异步图书
www.epubit.com

Pearson

重构
改善既有代码的设计
（第2版）

[美] 马丁·福勒（Martin Fowler）著
熊节 林从羽 译

REFACTORING
Improving the Design of Existing Code
SECOND EDITION



中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

目　　录

版权信息

书名：重构：改善既有代码的设计（第2版）

ISBN：978-7-115-50865-2

著　　　　[美] 马丁•福勒（Martin Fowler）

译　　　　熊　节　林从羽

责任编辑

人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

读者服务热线：(010)81055410

反盗版热线：(010)81055315

内容提要

本书是经典著作《重构》出版20年后的新版。书中清晰揭示了重构的过程，解释了重构的原理和最佳实践方式，并给出了何时以及何地应该开始挖掘代码以求改善。书中给出了60多个可行的重构，每个重构都介绍了一种经过验证的代码变换手法的动机和技术。本书提出的重构准则将帮助开发人员一次一小步地修改代码，从而减少了开发过程中的风险。

本书适合软件开发人员、项目管理人员等阅读，也可作为高等院校计算机及相关专业师生的参考读物。

版权声明

对本书的赞誉

过去20年，《重构》一直是我案头常备的图书。每次重读，仍有感悟。对我而言，《重构》的意义不只在于指导代码重构，更在于让人从一开始就知道什么是好的代码，并且尽量写出没有“坏味道”的代码。Martin Fowler这次对本书进行的重构，体现了近年来编程领域的一些思潮变化。看来，既有设计，永远有改进空间。

——韩磊，《代码整洁之道》译者

重构早就成了软件开发从业者本能的一部分，每个IDE都内置了重构功能，每个程序员都定期重构自己的代码。技能上通常不再是问题，但是相对于当年第1版的读者，现在的程序员对于重构这个思想从何而来以及各种细节反而更陌生，这时候就更值得重新读一下这本书了。

——霍炬，PRESS.one CTO

有人说Martin Fowler改变了人类开发软件的模式，这一点也不过分，从《分析模式》《UML精粹》《领域特定语言》，到这本《重构》新版可以

看得出来，他的每一本书都是软件开发人员必备的案头读物。此前他参与的“敏捷宣言”，更是引领了整个行业对敏捷开发的认识，一直到现在。Martin Fowler是我们QCon全球软件开发大会进入中国时的第一届讲师，也是在那次会议上，他让国内的技术社区领略了国际领先的开发模式，从此“敏捷”二字开始风行国内IT领域。

今年是QCon进入中国的第十个年头，我特别开心看到Martin Fowler又重写《重构》这本影响深远的书，他几乎完全替换了书中所引用的模式案例，并且基于现在用户的习惯，采用了JavaScript语言来做说明语言。数十年来他始终保持对技术的关注，对创新的热情，乐此不疲，这是Martin最令人敬佩的地方，也是非常值得我们每一个技术人学习的地方。

——霍泰稳，极客邦科技、InfoQ中国创始人兼CEO

当今软件开发的速度越来越快，带来的技术债也越来越多，我从CSDN自身的网站系统开发中充分认识到重构的重要性——如果我们的程序员能理解和掌握重构的原则和方法，我们的系统就不会有这么多沉重的债务。真正本质的东西是不变的，《重构》在出版20年后推出了第2版，再次证明：越本质的越长久，也越重要。衷心期待更多的新一

代开发者能从这本书吸收营养，开发出好味道的系统。

——蒋涛，CSDN创始人、董事长

最早看到本书第1版的英文原版并决定引进国内，算起来已经是20年前的事了。虽然时间是最强大的重构工具，连书里的示例语言都从Java变成JavaScript了，但书中的理念和实践的价值并没有随时间流逝。这充分证明，即使在日新月异的IT技术世界里，不变的东西其实还是有的，这种书才是真正的经典，是技术人员应该优先研读并一读再读的。

——刘江，美团技术学院院长

“对于软件工程师来说，重构，并不是额外的工作，它就是编码本身。”直到我读过《重构》，并经过练习，才真正理解到这一点。真希望自己在20多年前写第一个软件时，就能读到这本书，从而能节省出大量调试或重复研究代码的时间。20年过去了，《重构》这本书也根据当前软件设计及相关工具的发展进行了一部分修订，更加贴近当前的软件开发者。希望更多的软件工程师能够应用这一技术节省出更多的时间。

——乔梁，腾讯高级管理顾问、《持续交付2.0》作者

重构是一项被低估了的技术能力。说起来，重构就是“不改变外在行为，而提高代码质量”这么简简单单的一句话，但其带来的影响却非常深远：它使我们在解决问题时可以放心地“先做对，再做好”——这种思路本身就可以极大地简化问题；它使我们消除无谓的意气之争——“所谓好，就是更少的坏味道”。我由衷地认为，切实地读懂了《重构》的程序员，在能力上都会获得一个数量级的提升。

——徐昊，ThoughtWorks中国区技术总监

当我还是编程菜鸟，想写出漂亮的代码而不得门道的时候，《重构》这本书就告诉了我，其实高手的代码也不是一次书就的，只要按这本书里的方法去做，谁都能把代码写得那么好；当我还是职场新人，没来得及写出太多垃圾代码的时候，这本书就教会了我，应该去追求编写人能够读懂的而不是仅机器能够读懂的代码。多年以后的某时某刻，当你编码自信而敏捷，因代码清晰而受人尊重时，你会庆幸读过这本书，你也会有些遗憾，应该再早一点去读这本书。无论过去了多少年，这本书，一直值得推荐。

——阎华，京东7FRESH架构师

在大获成功的《重构》第1版里，Martin

Fowler传达的核心理念是：代码会随时间流逝而烂掉。写得再好的程序代码，若是发布了就一直保持原样，照样会风化、破碎乃至分崩离析。这是客观规律，避免这种命运的唯一出路是持续重构。要想成为高素质的软件工程师，必须认识这一点。

20年之后，Martin Fowler用现身说法证明，经典的《重构》也会变得不合时宜，也需要重构。如今，不但讲解语言从Java改成了JavaScript，原来的重构示例也做了很多调整，新增了15个示例，更重要的是，新版示例不再那么“面向对象”，应当会收获更广泛的读者群。

软件不死，重构不歇。

——余晟，《代码整洁之道：程序员的职业素养》译者

随着软件项目日积月累，系统维护成本变得越来越高昂是互联网团队共同面临的问题。用户在使用互联网系统的过程中，遇到的各类运行错误或者不可访问故障，以及开发团队面临的历史系统不可维护问题，很多时候是代码初次开发过程中各种细小的不规范引起的。持续优化已有代码是维护系统生命力最好的方法。《重构》是我推荐团队必读的技术图书之一。

——杨卫华（Tim Yang），微博研发副总经理

软件行业已经高速发展数十年，就好似一个崭新的城市，从一个个村屋矮房到高楼林立。而你的代码库就好比你手下的一个房间、一幢平房、一条街道、一片社区乃至是一座摩天大楼。作为一本经典的软件开发书，《重构》告诉我们的不仅仅是如何推倒重建、清理、装修，而是像一个规划师一样从目的、成本、手段、价值等综合维度来思考重构的意义。在开发业务的同时，《重构》常伴我左右，警醒我如何写出更有价值的软件。

——阴明，掘金社区创始人

重构，是一个优秀程序员的基本功，因为没人能保证其代码不随时间腐化，而重构会让代码重新焕发活力。整个软件行业对重构的认知始于Martin Fowler的《重构》，这本书让人们知道了“代码的坏味道”，见识到了“小步前行”的威力。时隔20年，Martin Fowler重新执笔改写《重构》，20年间的思维变迁就体现在这本书里，在第1版中，我们看到的是当时方兴未艾的面向对象，而第2版则透露出函数式编程的影响。如果说有什么程序员进阶秘笈，那就是不要错过Martin Fowler的任何一部著作，更何况是已经由时间证明过的重要著作《重构》的新版！

——郑晔，火币网首席架构师

如果看完本书，就兴冲冲地想要找一些代码来重构，那你可能就陷入某种“自嗨”之中了。

了解本书中列出的那些坏味道，不仅仅可以发现代码中的那些坏味道，更可以鞭策自己以及整个团队：在一开始的时候，就不写或者少些那种味道很坏的代码。还应该激励自己，深入地理解架构、理解业务、理解需求，减少因设计失误而导致徒劳无益地反复重构。

重构也是有成本的，所以应该思考如何降低重构的成本。我推荐每一个程序员都来学习“重构”这门手艺。因为学习《重构》，是为了减少“重构”！

——庄表伟，开源社理事、执行长，华为云DevCloud高级产品经理

重读《重构》，呼唤匠艺（译者序）

2009年，在为《重构》第1版的中译本再版整理译稿时，我已经隐约察觉行业中对“重构”这个概念的矛盾张力。一方面，在这个“VUCA”（易变、不确定、复杂、模糊）横行的年代，有能力调整系统的内部结构，使其更具长期生命力，这是一个令人神往的期许。另一方面，重构的扎实功夫要学起来、做起来，颇不是件轻松的事，且不说详尽到近乎琐碎的重构手法，光是单元测试一事，怕是已有九成同行无法企及。结果，“重构”渐渐成了一块漂亮的招牌，大家都愿意挂上这个名号，可实际上干的却多是“刀劈斧砍”的勾当。

如今又是10年过去，只从国内的情况而论，“重构”概念的表里分离，大有愈演愈烈之势。随着当年的一线技术人员纷纷走上领导岗位，他们乐于将“重构”这块漂亮招牌用在更宽泛的环境下，例如系统架构乃至组织结构，都可以“重构”一下。然而基本功的欠缺，却也一路如影随形。当年在对象中的刀劈斧砍，如今被照搬到了架构、组织的调整。于是“重构”的痛苦回忆又一遍遍重演，甚而程度更深、影响更广、为害更烈。

此时转头看Martin Fowler时隔将近廿载后终于付梓的《重构》第2版，我不禁感叹于他对“微末功夫”的执着。在此书尚未成型之前，我和当时ThoughtWorks的同事曾有很多猜测，猜Fowler先生是否会在第2版中拔高层次，多谈谈设计乃至架构级别的重构手法，甚或跟随“敏捷组织”“精益企业”的风潮谈谈组织重构，也未为不可。孰料成书令我们跌破眼镜，Fowler先生不仅没有拔高，反而把工夫做得更扎实了。

对比前后两版的重构列表，可以发现：第2版收录的重构手法在用途上更加内聚，在操作上更加连贯，更重视重构手法之间的组合运用。第1版中占了整章篇幅的“大型重构”，在第2版中全数删去。一些较为复杂的重构手法，例如复制“被监视数据”、塑造模板函数等，第2版也不再收录。而第2版中新增的重构手法，则多是提炼变量、移动语句、拆分循环、拆分变量这样更加细致而微的操作。这些新增的手法看似简单，但直指大规模遗留代码中最常见的重构难点，正好补上了第1版中阙漏的细节。这一变化，正反映出Fowler先生对于重构一事一贯的态度：千里之行积于跬步，越是面对复杂多变的外部环境，越是要做好基本功、迈出扎实步。

识别坏味道、测试先行、行为保持的变更动

作，是重构的基本功。在《重构》第2版里，重构手法的细节被再度打磨，重构过程比之第1版愈发流畅。细细品味重构手法中的前后步骤，琢磨作者是如何做到行为保持的，这是能启发读者举一反三的读书法。以保持对象完整重构手法为例，第1版中的做法是在原本函数上新添参数，而第2版的做法则是先新建一个空函数，在其中做完想要的调整之后，再整体替换原本函数。两相对比，无疑是新的做法更加可控、出错时测试失败的范围更小。

无独有偶，我在ThoughtWorks时的同事王健在开展大型的架构重构时，总结了重构的“十六字心法”，恰与保持对象完整重构手法在第2版中这个新的做法暗合。这十六字心法如是说：

旧的不变，

新的创建，

一步切换，

旧的再见。

从这个视角品味一个个重构巨细靡遗的做法，读者大概能感受到重构与“刀劈斧砍”之间最根本的分歧。在很多重构（例如最常用的改变函数声明）的做法中，Fowler先生会引入“很快就会再次修改甚至删除”的临时元素。假如只看起止状态，这些变

更过程中的临时元素似乎是浪费：为何不直接一步到位改变到完善的结果状态呢？然而这些临时元素所代表的，是对变更过程（而非只是结果）的设计。缺乏对过程的精心设计与必要投入，只抱着对结果的美好憧憬提刀上阵，遇到困难就靠“奋斗精神”和加班解决，这种“刀劈斧砍”不止发生在缺乏审慎的“重构”现场，又何尝不是我们这个行业的缩影？

是以，重构这门技艺，以及Fowler先生撰写《重构》的态度，代表的是软件开发的匠艺——对“正确的做事方式”的重视。在一个浮躁之风日盛的行业中，很多人会强调“只看结果”，轻视做事的过程与方式。然而对于软件开发的专业人士而言，如果忽视了过程与方式，也就等于放弃了我们自己的立身之本。Fowler先生近廿载对这本书、对重构手法的精心打磨，给了我们一个榜样：一个对匠艺上心的专业人士，日积月累对过程与方式的重视，是能有所成就的。

17年前，我以菜鸟之身读到《重构》，深受其中蕴涵的工匠精神感召，在Fowler先生与侯捷老师的帮助下，完成了本书第1版的翻译工作。如今再译本书第2版，来自ThoughtWorks的青年才俊林从羽君主动请缨与我搭档合译，我亦将此视为匠艺传承的一桩美事。新一代程序员中，关注新工具、新

框架、新商业模式者伙矣，关注面向对象、TDD、重构之类基本功者寥寥。林君年纪虽轻，却能平心静气磨砺技艺，对基本功学而时习，颇有老派工匠之风。当年藉由翻译《重构》一书，我从Fowler先生、侯捷老师身上学到他们的工匠精神，十余年来时时践行自勉。如今新一代软件工匠的代表人物林君接手此书，必会令工匠精神传承光大。

据说古时高僧有偈云："时时勤拂拭，勿使惹尘埃。"代码当如是，专业人士的技艺亦当如是。与《重构》的诸位读者共勉。

熊节

2019年1月26日于成都

译者简介

熊节　在IT行业已经打拼了18年，在金融、零售、政府、电信、制造业等行业的信息化建设方面有着丰富经验，是中国IT业敏捷浪潮的领军人物。熊节拥有利物浦大学MBA学位。



林从羽　ThoughtWorks软件开发工程师，曾服务于国内外多家大型企业，致力于帮助团队更快更好地交付可工作的软件。拥抱敏捷精神，TDD爱好者，纯键盘工作者。



第1版序

“重构”这个概念来自Smalltalk圈子，没多久就进入了其他编程语言阵营之中。因为重构是框架开发中不可缺少的一部分，所以当框架设计者讨论自己的工作时，这个术语就诞生了。当他们精炼自己的类继承体系时，当他们叫喊自己可以拿掉多少多少行代码时，重构的概念慢慢浮出水面。框架设计者知道，这东西不可能一开始就完全正确，它将随着设计者的经验成长而进化；他们也知道，代码被阅读和被修改的次数远远多于它被编写的次数。保持代码易读、易修改的关键，就是重构——对框架而言如此，对一般软件也如此。

好极了，还有什么问题吗？问题很显然：重构有风险。它必须修改正在工作的程序，这可能引入一些不易察觉的错误。如果重构方式不恰当，可能毁掉你数天甚至数周的成果。如果重构时不做好准备，不遵守规则，风险就更大。你挖掘自己的代码，很快发现了一些值得修改的地方，于是你挖得更深。挖得越深，找到的重构机会就越多，于是你的修改也越多……最后你给自己挖了个大坑，却爬不出去了。为了避免自掘坟墓，重构必须系统化进

行。我和三位合作者在写《设计模式》一书时曾经提过：设计模式为重构提供了目标。然而“确定目标”只是问题的一部分而已，改造程序以达到目标，是另一个难题。

Martin Fowler和本书另几位作者清楚地揭示了重构过程，他们为面向对象软件开发所做的贡献难以估量。本书解释了重构的原理和最佳实践，并指出何时何地你应该开始挖掘你的代码以求改善。本书的核心是一系列完整的重构方法，其中每一项都介绍一种经过实践检验的代码变换手法的动机和技术。某些项目（如“提炼函数”和“搬移字段”）看起来可能很浅显，但不要掉以轻心，因为理解这类技术正是有条不紊地进行重构的关键。本书所提的这些重构手法将帮助你一次一小步地修改你的代码，这就降低了设计演进过程中的风险。很快你就会把这些重构手法及其名称加入自己的开发词典中，并且朗朗上口。

我第一次体验有条不紊的、一次一小步的重构，是某次与Kent Beck在三万英尺高空的飞行旅途中结对编程。我们运用本书中收录的重构手法，保证每次只走一步。最后，我对这种实践方式的效果感到十分惊讶。我不但对产生的代码更有信心，而且开发压力也小了很多。因此，我极力推荐你试试这些重构手法，你和你的程序都将因此更美好。

——Erich Gamma

Object Technology International, Inc.

1999年1月

前言

从前，有位咨询顾问造访客户调研其开发项目。该系统的核心是一个类继承体系，顾问看了开发人员所写的一些代码。他发现整个体系相当凌乱，上层超类对系统的工作方式做了一些假设，下层子类实现这些假设。但是这些假设并不适合所有子类，导致覆写（override）工作非常繁重。只要在超类做点修改，就可以减少许多覆写工作。在另一些地方，超类的某些意图并未被良好理解，因此其中某些行为在子类内重复出现。还有一些地方，好几个子类做相同的事情，其实可以把它们搬到继承体系的上层去做。

这位顾问于是建议项目经理看看这些代码，把它们整理一下，但是项目经理并不热衷于此，毕竟程序看上去还可以运行，而且项目面临很大的进度压力。于是项目经理说，晚些时候再抽时间做这些整理工作。

顾问也把他的想法告诉了在这个继承体系上工作的程序员，告诉他们可能发生的事情。程序员都很敏锐，马上就看出问题的严重性。他们知道这并

不全是他们的错，有时候的确需要借助外力才能发现问题。程序员立刻用了一两天的时间整理好这个继承体系，并删掉了其中一半代码，功能毫发无损。他们对此十分满意，而且发现在继承体系中加入新的类或使用系统中的其他类都更快、更容易了。

项目经理并不高兴。进度排得很紧，有许多工作要做。系统必须在几个月之后发布，而这些程序员却白白耗费了两天时间，做的工作与未来几个月要交付的大量功能毫不相干。原先的代码运行起来还算正常。的确，新的设计更加“纯粹”、更加“整洁”。但项目要交付给客户的，是可以有效运行的代码，不是用以取悦学究的代码。顾问接下来又建议应该在系统的其他核心部分进行这样的整理工作，这会使整个项目停顿一至两个星期。所有这些工作只是为了让代码看起来更漂亮，并不能给系统添加任何新功能。

你对这个故事有什么感想？你认为这个顾问的建议（更进一步整理程序）是对的吗？你会遵循那句古老的工程谚语吗：“如果它还可以运行，就不要动它。”

我必须承认自己有某些偏见，因为我就是那个顾问。6个月之后这个项目宣告失败，很大的原因是代码太复杂，无法调试，也无法将性能调优到可

接受的水平。

后来，这个项目重新启动，几乎从头开始编写整个系统，Kent Beck受邀做了顾问。他做了几件迥异以往的事，其中最重要的一件就是坚持以持续不断的重构行为来整理代码。这个团队效能的提升，以及重构在其中扮演的角色，启发了我撰写本书的第1版，如此一来我就能够把Kent和其他一些人已经学会的“以重构方式改进软件质量”的知识，传播给所有读者。

自本书第1版问世至今，读者的反馈甚佳，重构的理念已经被广泛接纳，成为编程的词汇表中不可或缺的部分。然而，对于一本与编程相关的书而言，18年已经太漫长，因此我感到，是时候回头重新修订这本书了。我几乎重写了全书的每一页，但从其内涵而言，整本书又几乎没有改变。重构的精髓仍然一如既往，大部分关键的重构手法也大体不变。我希望这次修订能帮助更多的读者学会如何有效地进行重构。

什么是重构

所谓重构（refactoring）是这样一个过程：在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构。重构是一种经千锤百炼形成的有条不紊的程序整理方法，可以最大限度地减小整理过程中引入错误的概率。本质上说，重构就是在代码写好之后改进它的设计。

“在代码写好之后改进它的设计”这种说法有点儿奇怪。在软件开发的大部分历史时期，大部分人相信应该先设计而后编码：首先得有一个良好的设计，然后才能开始编码。但是，随着时间流逝，人们不断修改代码，于是根据原先设计所得的系统，整体结构逐渐衰弱。代码质量慢慢沉沦，编码工作从严谨的工程堕落为胡砍乱劈的随性行为。

“重构”正好与此相反。哪怕手上有一个糟糕的设计，甚至是一堆混乱的代码，我们也可以借由重构将它加工成设计良好的代码。重构的每个步骤都很简单，甚至显得有些过于简单：只需要把某个字段从一个类移到另一个类，把某些代码从一个函数拉出来构成另一个函数，或是在继承体系中把某些代码推上推下就行了。但是，聚沙成塔，这些小小

的修改累积起来就可以根本改善设计质量。这和一般常见的“软件会慢慢腐烂”的观点恰恰相反。

有了重构以后，工作的平衡点开始发生变化。我发现设计不是在一开始完成的，而是在整个开发过程中逐渐浮现出来。在系统构筑过程中，我学会了如何不断改进设计。这个“构筑-设计”的反复互动，可以让一个程序在开发过程中持续保有良好的设计。

本书有什么

本书是一本为专业程序员编写的重构指南。我的目的是告诉你如何以一种可控且高效的方式进行重构。你将学会如何有条不紊地改进程序结构，而且不会引入错误，这就是正确的重构方式。

按照传统，图书应该以概念介绍开头。尽管我也同意这个原则，但是我发现以概括性的讨论或定义来介绍重构，实在不是一件容易的事。因此，我决定用一个实例作为开路先锋。第1章展示了一个小程序，其中有些常见的设计缺陷，我把它重构得更容易理解和修改。其间你可以看到重构的过程，以及几个很有用的重构手法。如果你想知道重构到底是怎么回事，这一章不可不读。

第2章讨论重构的一般性原则、定义，以及进行重构的原因，我也大致介绍了重构面临的一些挑战。第3章由Kent Beck介绍如何嗅出代码中的“坏味道”，以及如何运用重构清除这些“坏味道”。测试在重构中扮演着非常重要的角色，第4章介绍如何在代码中构筑测试。

从第5章往后的篇幅就是本书的核心部分——重构名录。尽管不能说是一份巨细靡遗的列表，却

足以覆盖大多数开发者可能用到的关键重构手法。这份重构名录的源头是20世纪90年代后期我开始学习重构时的笔记，直到今天我仍然不时查阅这些笔记，作为对我不甚可靠的记忆力的补充。每当我想做点什么——例如拆分阶段（154）——的时候，这份列表就会提醒我如何一步一步安全前进。我希望这是值得你日后一再回顾的部分。

一本Web优先的书1

万维网对我们的社会影响深远，尤其是改变了我们获取信息的方式。在撰写本书第1版时，关于软件开发的知识大多通过出版物传播。而时至今日，我的大部分信息都来自网上。这个趋势给像我这样的写作者带来了一个挑战：今日世界还有图书的一席之地吗？今天的图书应该是什么形态？

我相信像这样一本书仍然有其价值，但也需要作出改变。图书的价值在于把大量信息以内聚的方式整合起来。在撰写本书的过程中，我尝试用连贯一致的方式来组织和涵盖大量各有特色的重构手法。

但这个聚合的整体是一个抽象的文学作品，尽管传统上只能以纸质图书的形式呈现，未来却未必

非得如此。出版行业仍然将纸质图书视为首要的呈现形式，虽然我们已经满怀热情地接纳了电子书，但是电子图书毕竟也只是在原来纸质图书结构的基础上做了电子化的呈现。

我想通过这本书探索一条不同的路径。本书的权威版本是它的网站（或者叫“Web版”）。如果你购买了纸质版或者电子版，就会同时获得访问Web版的权限。（关于如何在InformIT网站上注册你的商品，请留意下文的提示。）纸质版图书是网站内容的精选，并整理成适合印刷的形式。纸质版并不尝试包含网站上的所有重构手法，尤其是考虑到未来我很有可能在Web版中增加更多重构手法。与此相似，电子书又是Web版的另一个呈现，其中包含的重构手法列表可能与纸质版不同，毕竟电子书在售出之后也可以相对容易地更新和添加内容。

在写下这些文字时，我无从知晓你正在阅读的是在线Web版、手机上的电子书、纸质版图书还是别的什么超乎我想象的形式。我尽力写一本有用的书，不论你用什么形式来汲取其中的知识。

如果你想查看本书Web版（只有英文版），并及时获得内容更新和勘误，请到InformIT网站注册这本书。你需要首先打开

informit.com/register页面，登录你的InformIT账户（如果没有InformIT账户的话，需要先注册一个），然后（进入“Registered Products”标签）输入本书英文原版的ISBN“9780134757599”，单击“Submit”按钮。然后网站会向你提出一个与本书内容有关的问题，所以请确保纸质书或电子书就放在手边。成功注册以后，进入“Account”页面，打开“Digital Purchases”标签，单击本书标题下面的“Launch”按钮，就能看到本书的Web版。

For access to the web edition (English only) and updates or corrections as they become available, register your copy on the InformIT web site. To start the registration process, go to informit.com/register and log in (or create an account if you don’t have one). Enter 9780134757599 in the box labeled ISBN and click Submit. You will be asked a challenge question, so be sure to have your copy of the book available. After you’ve successfully registered your copy, open the “Digital Purchases” tab on your Account page and click on the link under this title to “Launch” the web edition.

JavaScript代码范例

与软件开发中的大多数技术性领域一样，代码范例对于概念的阐释至关重要。不过，即使在不同的编程语言中，重构手法看上去也是大同小异的。虽然会有一些值得留心的语言特性，但重构手法的核心要素都是一样的。

我选择了用JavaScript来展现本书中的重构手法，因为我感到大多数读者都能看懂这种语言。不过，即便你眼下正在使用的是别的编程语言，采用这些重构手法也应该不困难。我尽量不使用JavaScript任何复杂的特性，这样即便你对这门编程语言只有粗浅的了解，应该也能跟上重构的过程。另外，使用JavaScript展示重构手法，并不代表我推荐这门编程语言。

使用JavaScript展示代码范例，也不意味着本书介绍的技巧只适用于JavaScript。本书的第1版采用了Java，但很多从未写过任何Java代码的程序员也同样认为这些技巧很有用。我曾经尝试过用十多种不同的编程语言来呈现这些范例，以此展示重构手法的通用性，不过这对普通读者而言只会带来困惑。本书是为所有编程语言背景的程序员所作，除了阅读“范例”小节时需要一些基本的JavaScript知

识，本书的其余部分都不特定于任何具体的编程语言。我希望读者能汲取本书的内容，并将其应用于自己日常使用的编程语言。具体而言，我希望读者能先理解本书中的JavaScript范例代码，然后再将其适配到自己习惯的编程语言。

因此，除了在特殊情况下，当我谈到“类”“模块”“函数”等词汇时，我都按照它们在程序设计领域的一般含义来使用这些词，而不是以其在JavaScript语言模型中的特殊含义来使用。

我只把JavaScript用作一种示例语言，因此我也会尽量避免使用其他程序员可能不太熟悉的编程风格。这不是一本“用JavaScript进行重构”的书，而是一本关于重构的通用书籍，只是采用了JavaScript作为示例。有很多JavaScript特有的重构手法很有意思（如将回调重构成promise或async/await），但这些不是本书要讨论的内容。

谁该阅读本书

本书的目标读者是专业程序员，也就是那些以编写软件为生的人。书中的范例和讨论，涉及大量需要详细阅读和理解的代码。这些例子都用JavaScript写成，不过这些重构手法应该适用于大部分编程语言。为了理解书中的内容，读者需要有一定的编程经验，但需要的知识并不多。

本书的首要目标读者群是想要学习重构的软件开发者，同时对于已经理解重构的人也有价值——本书可以作为一本教学辅助书。在本书中，我用了大量篇幅详细解释各个重构手法的过程和原理，因此有经验的开发人员可以用本书来指导同事。

尽管本书的关注对象是代码，但重构对于系统设计也有巨大影响。资深设计师和架构师也很有必要了解重构原理，并在自己的项目中运用重构技术。最好是由有威望的、经验丰富的开发人员来引入重构技术，因为这样的人最能够透彻理解重构背后的原理，并根据情况加以调整，使之适用于特定工作领域。如果你使用的不是JavaScript而是其他编程语言，这一点尤其重要，因为你必须把我给出的范例用其他编程语言改写。

下面我要告诉你，如何能够在不通读全书的情况下充分用好它。

- 如果你想知道重构是什么，请阅读第1章，其中的示例会让你弄清楚重构的过程。
- 如果你想知道为什么应该重构，请阅读前两章，它们会告诉你重构是什么以及为什么应该重构。
- 如果你想知道该在什么地方重构，请阅读第3章，它会告诉你一些代码特征，这些特征指出“这里需要重构”。
- 如果你想着手进行重构，请完整阅读前四章，然后选择性地阅读重构名录。一开始只需概略浏览列表，看看其中有些什么，不必理解所有细节。一旦真正需要实施某个重构手法，再详细阅读它，从中获取帮助。列表部分是供查阅的参考性内容，你不必一次就把它全部读完。

给形形色色的重构手法命名是编写本书的重要部分。合适的词汇能帮助我们彼此沟通。当一名开发者向另一名开发者提出建议，将一段代码提取成为一个函数，或者将计算逻辑拆分成几个阶段，双方都能理解提炼函数（106）和拆分阶段（154）是什么意思。这份词汇表也能帮助开发者选择自动化

的重构手法。

[1]这一节中关于各个版本的表述仅适用于本书的英文原版，中文版的相关版本可能会与此略有不同。——编者注

站在前人的肩膀上

就在本书一开始的此时此刻，我必须说：这本书让我欠了一大笔人情债，欠那些在20世纪90年代做了大量研究工作并开创重构领域的人一大笔债。学习他们的经验启发了我撰写本书第1版，尽管已经过去了很多年，我仍然必须感谢他们打下的基础。这本书原本应该由他们之中的某个人来写，但最后却让我这个有时间、有精力的人捡了便宜。

重构技术的两位最早倡导者是**Ward Cunningham**和Kent Beck。他们很早就把重构作为软件开发过程的一块基石，并且在自己的开发过程中运用它。尤其需要说明的是，正因为和Kent合作，我才真正看到了重构的重要性，并直接受到激励写了这本书。

Ralph Johnson在UIUC（伊利诺伊大学厄巴纳-香槟分校）领导了一个小组，这个小组因其在对象技术方面的实用贡献而声名远扬。Ralph很早就是重构的拥护者，他的一些学生也在重构领域的发展前期做出重要研究。Bill Opdyke的博士论文是重构研究的第一份详细的书面成果。John Brant和Don

Roberts则早已不满足于写文章了，他们创造了第一个自动化的重构工具，这个叫作Refactoring Browser（重构浏览器）的工具可以用于重构Smalltalk程序。

自本书第1版问世以来，很多人推动了重构领域的发展。尤其是，开发工具中的自动化重构功能，让程序员的生活轻松了许多。如今我只要简单地敲几下键盘就可以给一个被大量使用的函数改名，对此我已经习以为常，但在这快捷的操作背后，离不开IDE开发团队的辛勤劳动。

致谢

尽管有这些研究成果可以借鉴，我还是需要很多协助才能写成本书。本书的第1版极大地得益于Kent Beck的经验与鼓励。起初向我介绍重构的是他，鼓励我开始书面记录重构手法的是他，帮助我把重构手法组织成型的也是他，提出“代码味道”这个概念的还是他。我常常感觉，他本可以把本书的第1版写得更好——如果当时他不是在忙着撰写极限编程的奠基之作《解析极限编程》的话。

我认识的所有技术图书作者都会提到，技术审稿人提供了巨大的帮助。我们的作品都会有巨大的缺陷，只有同行审稿人能发现这些缺陷。我自己并不常做技术审稿，部分原因是我认为自己并不擅长，所以我对优秀的技术审稿人总是满怀敬意。帮别人审稿所得的报酬微不足道，所以这完全是一项慷慨之举。

正式开始写这本书时，我建了一个邮件列表，其中都是能给我提供反馈的建议者。随着写作的进展，我不断把新的草稿发到这个小组里，请他们给我反馈。我要感谢这些人在邮件列表中提供的反馈：Arlo Belshee、Avdi Grimm、Beth Anders-

Beck、Bill Wake、Brian Guthrie、Brian Marick、Chad Wathington、Dave Farley、David Rice、Don Roberts、Fred George、Giles Alexander、Greg Doench、Hugo Corbucci、Ivan Moore、James Shore、Jay Fields、Jessica Kerr、Joshua Kerievsky、Kevlin Henney、Luciano Ramalho、Marcos Brizeno、Michael Feathers、Patrick Kua、Pete Hodgson、Rebecca Parsons和Trisha Gee。

在这群人中，我要特别感谢Beth Anders-Beck、James Shore和Pete Hodgson在JavaScript方面给我的帮助。

有了一个比较完整的初稿之后，我将它发送出去，寻求更多的审阅意见，因为我希望有一些全新的眼光来纵览全书。William Chargin和Michael Hunger提供了极其详尽的审阅意见。我还从Bob Martin和Scott Davis那里得到了很多有用的意见。Bill Wake也对本书初稿做了完整的审阅，并在邮件列表中给出了他的意见。

我在ThoughtWorks的同事一直给我的写作提供想法和反馈。数不胜数的问题、评论和观点推动了本书的思考与写作。作为ThoughtWorks员工最好的一件事，就是这家公司允许我花大量时间来写作。我尤其要感谢Rebecca Parsons（我们的CTO）经常与我交流，给了我很多想法。

在培生出版集团，Greg　Doench是负责本书的策划编辑，他解决了无数的问题，最终使本书得以出版；Julie Nahil是责任编辑；Dmitry Kirsanov负责文字编辑工作；Alina　Kirsanova负责排版和制作索引。我也很高兴与他们合作。

服务与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，点击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的100积分。积分可用于在异步社区兑换优惠券、样书或奖品。


详细信息
写书评
提交勘误
页码:
页内位置（行数）:
勘误印次:
字数统计
提交

与我们联系

我们的联系邮箱是contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问www.epubit.com/selfpublish/submission即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下IT专业图书社区，致力于出版精品IT技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于2015年8月，提供大量精品IT技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品IT专业图书的品牌，依托于人民邮电出版社近30年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

微信服务号

第1章　重构，第一个示例

我该从何说起呢？按照传统做法，一开始介绍某样东西时应该先大致讲讲它的历史、主要原理等。可是每当有人在会场上介绍这些东西，总是诱发我的瞌睡虫。我的思绪开始游荡，我的眼神开始迷离，直到主讲人秀出示例，我才能够提起精神。

示例之所以可以拯救我于太虚之中，因为它让我看见事情在真正进行。谈原理，很容易流于泛泛，又很难说明如何实际应用。给出一个示例，就可以帮助我把事情认识清楚。

因此，我决定从一个示例说起。在此过程中我会谈到很多重构的工作方式，并且让你对重构过程有一点点感觉。然后在下一章中我才能向你展开通常的原理介绍。

但是，面对这个介绍性示例，我遇到了一个大问题。如果我选择一个大型程序，那么对程序自身的描述和对整个重构过程的描述就太复杂了，任何读者都不忍卒读（我试了一下，哪怕稍微复杂一点的例子都会超过100页）。如果我选择一个容易理

解的小程序，又恐怕看不出重构的价值。

和任何立志要介绍“应用于真实世界的程序中的有用技术”的人一样，我陷入了一个十分典型的两难困境。我只能带你看看如何在一个我选择的小程序中进行重构，然而坦白说，那个程序的规模根本不值得我们这么做。但是，如果我给你看的代码是大系统的一部分，重构技术很快就变得重要起来。所以请你一边观赏这个小例子，一边想象它身处于一个大得多的系统。

1.1　起点

在本书第1版中，我使用的示例程序是为影片出租店的顾客打印一张详单。放到今天，很多人可能要问了："影片出租店是什么？"为了避免过多回答这个问题，我翻新了一下示例，将其包装成一个仍有古典韵味又尚未消亡的现代示例。

设想有一个戏剧演出团，演员们经常要去各种场合表演戏剧。通常客户（customer）会指定几出剧目，而剧团则根据观众（audience）人数及剧目类型来向客户收费。该团目前出演两种戏剧：悲剧（tragedy）和喜剧（comedy）。给客户发出账单时，剧团还会根据到场观众的数量给出"观众量积分"（volume credit）优惠，下次客户再请剧团表演时可以使用积分获得折扣——你可以把它看作一种提升客户忠诚度的方式。

该剧团将剧目的数据存储在一个简单的JSON文件中。

plays.json...

```
{
```

```
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

他们开出的账单也存储在一个JSON文件里。

invoices.json...

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
    ]
  }
]
```

下面这个简单的函数用于打印账单详情。

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
```

```
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(p

    // print line for this order
    result += ` ${play.name}: ${format(thisAmount/100)} (${pe
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

用上面的数据文件（invoices.json和plays.json）作为测试输入，运行这段代码，会得到如下输出：

```
Statement for BigCo
  Hamlet: $650.00 (55 seats)
  As You Like It: $580.00 (35 seats)
  Othello: $500.00 (40 seats)
Amount owed is $1,730.00
You earned 47 credits
```

1.2　对此起始程序的评价

你觉得这个程序设计得怎么样？我的第一感觉是，代码组织不甚清晰，但这还在可忍受的限度内。这样小的程序，不做任何深入的设计，也不会太难理解。但我前面讲过，这是因为要保证例子足够小的缘故。如果这段代码身处于一个更大规模——也许是几百行——的程序中，把所有代码放到一个函数里就很难理解了。

尽管如此，这个程序还是能正常工作。那么是不是说，对其结构“不甚清晰”的评价只是美学意义上的判断，只是对所谓丑陋代码的反感呢？毕竟编译器也不会在乎代码好不好看。但是，当我们需要修改系统时，就涉及了人，而人在乎这些。差劲的系统是很难修改的，因为很难找到修改点，难以了解做出的修改与现有代码如何协作实现我想要的行为。如果很难找到修改点，我就很有可能犯错，从而引入bug。

因此，如果我需要修改一个有几百行代码的程序，我会期望它有良好的结构，并且已经被分解成一系列函数和其他程序要素，这能帮我更易于清楚

地了解这段代码在做什么。如果程序杂乱无章，先为它整理出结构来，再做需要的修改，通常来说更加简单。

> 如果你要给程序添加一个特性，但发现代码因缺乏良好的结构而不易于进行更改，那就先重构那个程序，使其比较容易添加该特性，然后再添加该特性。

在这个例子里，我们的用户希望对系统做几个修改。首先，他们希望以HTML格式输出详单。现在请你想一想，这个变化会带来什么影响。对于每处追加字符串到`result`变量的地方我都得为它们添加分支逻辑。这会为函数引入更多复杂度。遇到这种需求时，很多人会选择直接复制整个方法，在其中修改输出HTML的部分。复制一遍代码似乎不算太难，但却给未来留下各种隐患：一旦计费逻辑发生变化，我就得同时修改两个地方，以保证它们逻辑相同。如果你编写的是一个永不需要修改的程序，这样剪剪贴贴就还好。但如果程序要保存很长时间，那么重复的逻辑就会造成潜在的威胁。

现在，第二个变化来了：演员们尝试在表演类型上做更多突破，无论是历史剧、田园剧、田园喜

剧、田园史剧、历史悲剧还是历史田园悲喜剧，无论一成不变的正统戏，还是千变万幻的新派戏，他们都希望有所尝试，只是还没有决定试哪种以及何时试演。这对戏剧场次的计费方式、积分的计算方式都有影响。作为一个经验丰富的开发者，我可以肯定：不论最终提出什么方案，他们一定会在6个月之内再次修改它。毕竟，需求通常不来则已，一来便会接踵而至。

为了应对分类规则和计费规则的变化，程序必须对statement函数做出修改。但如果我把statement内的代码复制到用以打印HTML详单的函数中，就必须确保将来的任何修改在这两个地方保持一致。随着各种规则变得越来越复杂，适当的修改点将越来越难找，不犯错的机会也越来越少。

我再强调一次，是需求的变化使重构变得必要。如果一段代码能正常工作，并且不会再被修改，那么完全可以不去重构它。能改进之当然很好，但若没人需要去理解它，它就不会真正妨碍什么。如果确实有人需要理解它的工作原理，并且觉得理解起来很费劲，那你就需要改进一下代码了。

1.3　重构的第一步

每当我要进行重构的时候，第一个步骤永远相同：我得确保即将修改的代码拥有一组可靠的测试。这些测试必不可少，因为尽管遵循重构手法可以使我避免绝大多数引入bug的情形，但我毕竟是人，毕竟有可能犯错。程序越大，我的修改不小心破坏其他代码的可能性就越大——在数字时代，软件的名字就是脆弱。

`statement`函数的返回值是一个字符串，我做的就是创建几张新的账单（invoice），假设每张账单收取了几出戏剧的费用，然后使用这几张账单作为输入调用`statement`函数，生成对应的对账单（statement）字符串。我会拿生成的字符串与我已经手工检查过的字符串做比对。我会借助一个测试框架来配置好这些测试，只要在开发环境中输入一行命令就可以把它们运行起来。运行这些测试只需几秒钟，所以你会看到我经常运行它们。

测试过程中很重要的一部分，就是测试程序对于结果的报告方式。它们要么变绿，表示所有新字符串都和参考字符串一样，要么就变红，然后列出失败清单，显示问题字符串的出现行号。这些测试

都能够自我检验。使测试能自我检验至关重要，否则就得耗费大把时间来回比对，这会降低开发速度。现代的测试框架都提供了丰富的设施，支持编写和运行能够自我检验的测试。

重构前，先检查自己是否有一套可靠的测试集。这些测试必须有自我检验能力。

进行重构时，我需要依赖测试。我将测试视为bug检测器，它们能保护我不被自己犯的错误所困扰。把我想要达成的目标写两遍——代码里写一遍，测试里再写一遍——我就得犯两遍同样的错误才能骗过检测器。这降低了我犯错的概率，因为我对工作进行了二次确认。尽管编写测试需要花费时间，但却为我节省下可观的调试时间。构筑测试体系对重构来说实在太重要了，因此我将用第4章一整章的笔墨来详细讨论它。

1.4　分解statement函数

每当看到这样长长的函数，我便下意识地想从整个函数中分离出不同的关注点。第一个引起我注意的就是中间那段switch语句。

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
```

```
    if ("comedy" === play.type) volumeCredits += Math.floor(p

    // print line for this order
    result += ` ${play.name}: ${format(thisAmount/100)} (${pe
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

看着这块代码，我就知道它在计算一场戏剧演出的费用。这是我的直觉。不过正如Ward Cunningham所说，这种理解只是我脑海中转瞬即逝的灵光。我需要梳理这些灵感，将它们从脑海中搬回到代码里去，以免忘记。这样当我回头看时，代码就能告诉我它在干什么，我不需要重新思考一遍。

要将我的理解转化到代码里，得先将这块代码抽取成一个独立的函数，按它所干的事情给它命名，比如叫`amountFor(performance)`。每次想将一块代码抽取成一个函数时，我都会遵循一个标准流程，最大程度减少犯错的可能。我把这个流程记录了下来，并将它命名为提炼函数（106），以便日后可以方便地引用。

首先，我需要检查一下，如果我将这块代码提炼到自己的一个函数里，有哪些变量会离开原本的作用域。在此示例中，是`perf`、`play`和`thisAmount`

这3个变量。前两个变量会被提炼后的函数使用，但不会被修改，那么我就可以将它们以参数方式传递进来。我更关心那些会被修改的变量。这里只有唯一一个——thisAmount，因此可以将它从函数中直接返回。我还可以将其初始化放到提炼后的函数里。修改后的代码如下所示。

function statement...

```
function amountFor(perf, play) {
  let thisAmount = 0;
  switch (play.type) {
  case "tragedy":
    thisAmount = 40000;
    if (perf.audience > 30) {
      thisAmount += 1000 * (perf.audience - 30);
    }
    break;
  case "comedy":
    thisAmount = 30000;
    if (perf.audience > 20) {
      thisAmount += 10000 + 500 * (perf.audience - 20);
    }
    thisAmount += 300 * perf.audience;
    break;
  default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return thisAmount;
}
```

当我在代码块上方使用了斜体（中文对应为楷体）标记的题头“*function xxx*”时，表明该代码块位

于题头所在函数、文件或类的作用域内。通常该作用域内还有其他的代码，但由于不是讨论重点，因此把它们隐去不展示。

现在原statement函数可以直接调用这个新函数来初始化thisAmount。

顶层作用域…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = amountFor(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(p

    // print line for this order
    result += ` ${play.name}: ${format(thisAmount/100)} (${pe
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

做完这个改动后，我会马上编译并执行一遍测试，看看有无破坏了其他东西。无论每次重构多么

简单，养成重构后即运行测试的习惯非常重要。犯错误是很容易的——至少我知道我是很容易犯错的。做完一次修改就运行测试，这样在我真的犯了错时，只需要考虑一个很小的改动范围，这使得查错与修复问题易如反掌。这就是重构过程的精髓所在：小步修改，每次修改后就运行测试。如果我改动了太多东西，犯错时就可能陷入麻烦的调试，并为此耗费大把时间。小步修改，以及它带来的频繁反馈，正是防止混乱的关键。

这里我使用的“编译”一词，指的是将JavaScript变为可执行代码之前的所有步骤。虽然JavaScript可以直接执行，有时可能不需任何步骤，但有时可能需要将代码移动到一个输出目录，或使用Babel这样的代码处理器等。

因为是JavaScript，我可以直接将`amountFor`提炼成为`statement`的一个内嵌函数。这个特性十分有用，因为我就不需要再把外部作用域中的数据传给新提炼的函数。这个示例中可能区别不大，但也是少了一件要操心的事。

重构技术就是以微小的步伐修改程

序。如果你犯下错误，很容易便可发现它。

做完上面的修改，测试是通过的，因此下一步我要把代码提交到本地的版本控制系统。我会使用诸如git或mercurial这样的版本控制系统，因为它们可以支持本地提交。每次成功的重构后我都会提交代码，如果待会不小心搞砸了，我便能轻松回滚到上一个可工作的状态。把代码推送（push）到远端仓库前，我会把零碎的修改压缩成一个更有意义的提交（commit）。

提炼函数（106）是一个常见的可自动完成的重构。如果我是用Java编程，我会本能地使用IDE的快捷键来完成这项重构。在我撰写本书时，JavaScript工具对此重构的支持仍不是很健壮，因此我必须手动重构。这不是很难，当然我还是需要小心处理那些局部作用域的变量。

完成提炼函数（106）手法后，我会看看提炼出来的函数，看是否能进一步提升其表达能力。一般我做的第一件事就是给一些变量改名，使它们更简洁，比如将`thisAmount`重命名为`result`。

function statement...

```
function amountFor(perf, play) {
```

```
  let result = 0;
  switch (play.type) {
  case "tragedy":
    result = 40000;
    if (perf.audience > 30) {
      result += 1000 * (perf.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (perf.audience > 20) {
      result += 10000 + 500 * (perf.audience - 20);
    }
    result += 300 * perf.audience;
    break;
  default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return result;
}
```

这是我个人的编码风格：永远将函数的返回值命名为“result”，这样我一眼就能知道它的作用。然后我再次编译、测试、提交代码。接着，我前往下一个目标——函数参数。

function statement...

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
```

```
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return result;
}
```

这是我的另一个编码风格。使用一门动态类型语言（如JavaScript）时，跟踪变量的类型很有意义。因此，我为参数取名时都默认带上其类型名。一般我会使用不定冠词修饰它，除非命名中另有解释其角色的相关信息。这个习惯是从Kent Beck那里学的[Beck SBPP]，到现在我还一直觉得很有用。

> 傻瓜都能写出计算机可以理解的代码。唯有能写出人类容易理解的代码的，才是优秀的程序员。

这次改名是否值得我大费周章呢？当然值得。好代码应能清楚地表明它在做什么，而变量命名是代码清晰的关键。只要改名能够提升代码的可读性，那就应该毫不犹豫去做。有好的查找替换工具在手，改名通常并不困难；此外，你的测试以及语

言本身的静态类型支持，都可以帮你揪出漏改的地方。如今有了自动化的重构工具，即便要给一个被大量调用的函数改名，通常也不在话下。

本来下一个要改名的变量是play，但我对这个参数另有安排。

移除play变量

观察amountFor函数时，我会看看它的参数都从哪里来。aPerformance是从循环变量中来，所以自然每次循环都会改变，但play变量是由performance变量计算得到的，因此根本没必要将它作为参数传入，我可以在amountFor函数中重新计算得到它。当我分解一个长函数时，我喜欢将play这样的变量移除掉，因为它们创建了很多具有局部作用域的临时变量，这会使提炼函数更加复杂。这里我要使用的重构手法是以查询取代临时变量（178）。

我先从赋值表达式的右边部分提炼出一个函数来。

function statement...

```
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
```

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(p

    // print line for this order
    result += ` ${play.name}: ${format(thisAmount/100)} (${pe
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

编译、测试、提交，然后使用内联变量（123）手法内联play变量。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, playFor(perf));

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Mat

    // print line for this order
    result += ` ${playFor(perf).name}: ${format(thisAmount/10
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

编译、测试、提交。完成变量内联后，我可以对amountFor函数应用改变函数声明（124），移除play参数。我会分两步走。首先在amountFor函数内部使用新提炼的函数。

function statement...

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (playFor(aPerformance).type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
```

```
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${playFor(aPerformance).
  }
  return result;
}
```

编译、测试、提交，最后将参数删除。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                      { style: "currency", currency: "USD",
                        minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    let thisAmount = amountFor(perf , playFor(perf) );

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Mat

    // print line for this order
    result += ` ${playFor(perf).name}: ${format(thisAmount/10
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
```

```
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

function statement...

```
function amountFor(aPerformance , play ) {
  let result = 0;
  switch (playFor(aPerformance).type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${playFor(aPerformance).
  }
  return result;
}
```

然后再一次编译、测试、提交。

这次重构可能在一些程序员心中敲响警钟：重构前，查找`play`变量的代码在每次循环中只执行了1次，而重构后却执行了3次。我会在后面探讨重构与性能之间的关系，但现在，我认为这次改动还不

太可能对性能有严重影响，即便真的有所影响，后续再对一段结构良好的代码进行性能调优，也容易得多。

移除局部变量的好处就是做提炼时会简单得多，因为需要操心的局部作用域变少了。实际上，在做任何提炼前，我一般都会先移除局部变量。

处理完amountFor的参数后，我回过头来看一下它的调用点。它被赋值给一个临时变量，之后就不再被修改，因此我又采用内联变量（123）手法内联它。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
    for (let perf of invoice.performances) {

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Mat

    // print line for this order
    result += ` ${playFor(perf).name}: ${format(amountFor(per
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
```

```
  return result;
```

提炼计算观众量积分的逻辑

现在statement函数的内部实现是这样的。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Mat

    // print line for this order
    result += ` ${playFor(perf).name}: ${format(amountFor(per
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

这会儿我们就看到了移除play变量的好处，移

除了一个局部作用域的变量，提炼观众量积分的计算逻辑又更简单一些。

我仍需要处理其他两个局部变量。perf同样可以轻易作为参数传入，但volumeCredits变量则有些棘手。它是一个累加变量，循环的每次迭代都会更新它的值。因此最简单的方式是，将整块逻辑提炼到新函数中，然后在新函数中直接返回volumeCredits。

function statement...

```
function volumeCreditsFor(perf) {
  let volumeCredits = 0;
  volumeCredits += Math.max(perf.audience - 30, 0);
  if ("comedy" === playFor(perf).type) volumeCredits += Math.
  return volumeCredits;
}
```

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
```

```
    // print line for this order
    result += ` ${playFor(perf).name}: ${format(amountFor(per
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

我还顺便删除了多余（并且会引起误解）的注释。

编译、测试、提交，然后对新函数里的变量改名。

function statement...

```
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === playFor(aPerformance).type) result += Math
  return result;
}
```

这里我只展示了一步到位的改名结果，不过实际操作时，我还是一次只将一个变量改名，并在每次改名后执行编译、测试、提交。

移除format变量

我们再看一下statement这个主函数。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                      { style: "currency", currency: "USD",
                        minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // print line for this order
    result += ` ${playFor(perf).name}: ${format(amountFor(per
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

正如我上面所指出的，临时变量往往会带来麻烦。它们只在对其进行处理的代码块中有用，因此临时变量实质上是鼓励你写长而复杂的函数。因此，下一步我要替换掉一些临时变量，而最简单的莫过于从format变量入手。这是典型的“将函数赋值给临时变量”的场景，我更愿意将其替换为一个明确声明的函数。

function statement...

```
function format(aNumber) {
  return new Intl.NumberFormat("en-US",
                    { style: "currency", currency: "USD",
                      minimumFractionDigits: 2 }).format(aN
}
```

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // print line for this order
    result += ` ${playFor(perf).name}: ${format(amountFor(per
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

尽管将函数变量改变成函数声明也是一种重构手法，但我既未为此手法命名，也未将它纳入重构名录。还有很多的重构手法我都觉得没那么重要。我觉得上面这个函数改名的手法既十分简单又不太常用，不值得在重构名录中占有一席之地。

我对提炼得到的函数名称不很满意——format未能清晰地描述其作用。formatAsUSD很表意，但又太长，特别它仅是小范围地被用在一个字符串模板中。我认为这里真正需要强调的是，它格式化的是一个货币数字，因此我选取了一个能体现此意图的命名，并应用了改变函数声明（124）手法。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // print line for this order
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

function statement...

```
function usd(aNumber) {
  return new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format(aN
}
```

好的命名十分重要，但往往并非唾手可得。只有恰如其分地命名，才能彰显出将大函数分解成小函数的价值。有了好的名称，我就不必通过阅读函数体来了解其行为。但要一次把名取好并不容易，因此我会使用当下能想到最好的那个。如果稍后想到更好的，我就会毫不犹豫地换掉它。通常你需要花几秒钟通读更多代码，才能发现最好的名称是什么。

重命名的同时，我还将重复的除以100的行为也搬移到函数里。将钱以美分为单位作为正整数存储是一种常见的做法，可以避免使用浮点数来存储货币的小数部分，同时又不影响用数学运算符操作它。不过，对于这样一个以美分为单位的整数，我又需要以美元为单位进行展示，因此让格式化函数来处理整除的事宜再好不过。

移除观众量积分总和

我的下一个重构目标是`volumeCredits`。处理这个变量更加微妙，因为它是在循环的迭代过程中累加得到的。第一步，就是应用拆分循环（227）将`volumeCredits`的累加过程分离出来。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {

    // print line for this order
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
    totalAmount += amountFor(perf);
  }
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }

result += `Amount owed is ${usd(totalAmount)}\n`;
result += `You earned ${volumeCredits} credits\n`;
return result;
```

完成这一步，我就可以使用移动语句（223）手法将变量声明挪动到紧邻循环的位置。

top level…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // print line for this order
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
    totalAmount += amountFor(perf);
  }
  let volumeCredits = 0;
```

```
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

把与更新volumeCredits变量相关的代码都集中到一起，有利于以查询取代临时变量（178）手法的施展。第一步同样是先对变量的计算过程应用提炼函数（106）手法。

function statement...

```
function totalVolumeCredits() {
  let volumeCredits = 0;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }
  return volumeCredits;
}
```

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // print line for this order
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
```

```
    totalAmount += amountFor(perf);
  }
  let volumeCredits = totalVolumeCredits();
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

完成函数提炼后，我再应用内联变量（123）手法内联`totalVolumeCredits`函数。

顶层作用域...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // print line for this order
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

重构至此，让我先暂停一下，谈谈刚刚完成的修改。首先，我知道有些读者会再次对此修改可能带来的性能问题感到担忧，我知道很多人本能地警惕重复的循环。但大多数时候，重复一次这样的循环对性能的影响都可忽略不计。如果你在重构前后进行计时，很可能甚至都注意不到运行速度的变化

——通常也确实没什么变化。许多程序员对代码实际的运行路径都所知不足，甚至经验丰富的程序员有时也未能避免。在聪明的编译器、现代的缓存技术面前，我们很多直觉都是不准确的。软件的性能通常只与代码的一小部分相关，改变其他的部分往往对总体性能贡献甚微。

当然，“大多数时候”不等同于“所有时候”。有时，一些重构手法也会显著地影响性能。但即便如此，我通常也不去管它，继续重构，因为有了一份结构良好的代码，回头调优其性能也容易得多。如果我在重构时引入了明显的性能损耗，我后面会花时间进行性能调优。进行调优时，可能会回退我早先做的一些重构——但更多时候，因为重构我可以使用更高效的调优方案。最后我得到的是既整洁又高效的代码。

因此对于重构过程的性能问题，我总体的建议是：大多数情况下可以忽略它。如果重构引入了性能损耗，先完成重构，再做性能优化。

其次，我希望你能注意到：我们移除`volumeCredits`的过程是多么小步。整个过程一共有4步，每一步都伴随着一次编译、测试以及向本地代码库的提交：

- 使用拆分循环（227）分离出累加过程；

- 使用移动语句（223）将累加变量的声明与累加过程集中到一起；
- 使用提炼函数（106）提炼出计算总数的函数；
- 使用内联变量（123）完全移除中间变量。

我得坦白，我并非总是如此小步——但在事情变复杂时，我的第一反应就是采用更小的步子。怎样算变复杂呢，就是当重构过程有测试失败而我又无法马上看清问题所在并立即修复时，我就会回滚到最后一次可工作的提交，然后以更小的步子重做。这得益于我如此频繁地提交。特别是与复杂代码打交道时，细小的步子是快速前进的关键。

接着我要重复同样的步骤来移除`totalAmount`。我以拆解循环开始（编译、测试、提交），然后下移累加变量的声明语句（编译、测试、提交），最后再提炼函数。这里令我有点头疼的是：最好的函数名应该是`totalAmount`，但它已经被变量名占用，我无法起两个同样的名字。因此，我在提炼函数时先给它随便取了一个名字（然后编译、测试、提交）。

function statement...

```
function appleSauce() {
```

```
  let totalAmount = 0;
  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }
  return totalAmount;
}
```

顶层作用域...

```
function statement (invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
  }
  let totalAmount = appleSauce();

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

接着我将变量内联（编译、测试、提交），然后将函数名改回`totalAmount`（编译、测试、提交）。

顶层作用域...

```
function statement (invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
```

```
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

function statement...

```
function totalAmount() {
  let totalAmount = 0;
  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }
  return totalAmount;
}
```

趁着给新提炼的函数改名的机会，我顺手一并修改了函数内部的变量名，以便保持我一贯的编码风格。

function statement...

```
function totalAmount() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += amountFor(perf);
  }
  return result;
}
function totalVolumeCredits() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
```

```
}
```

1.5 进展：大量嵌套函数

重构至此，是时候停下来欣赏一下代码的全貌了。

```
function statement (invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

  function totalAmount() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += amountFor(perf);
    }
    return result;
  }
  function totalVolumeCredits() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += volumeCreditsFor(perf);
    }
    return result;
  }
  function usd(aNumber) {
    return new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format(
  }
  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
```

```
    if ("comedy" === playFor(aPerformance).type) result += Ma
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID];
  }
  function amountFor(aPerformance) {
    let result = 0;
    switch (playFor(aPerformance).type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`unknown type: ${playFor(aPerformance).
    }
    return result;
  }
}
```

现在代码结构已经好多了。顶层的statement函数现在只剩7行代码，而且它处理的都是与打印详单相关的逻辑。与计算相关的逻辑从主函数中被移走，改由一组函数来支持。每个单独的计算过程和详单的整体结构，都因此变得更易理解了。

1.6　拆分计算阶段与格式化阶段

到目前为止，我的重构主要是为原函数添加足够的结构，以便我能更好地理解它，看清它的逻辑结构。这也是重构早期的一般步骤。把复杂的代码块分解为更小的单元，与好的命名一样都很重要。现在，我可以更多关注我要修改的功能部分了，也就是为这张详单提供一个HTML版本。不管怎么说，现在改起来更加简单了。因为计算代码已经被分离出来，我只需要为顶部的7行代码实现一个HTML的版本。问题是，这些分解出来的函数嵌套在打印文本详单的函数中。无论嵌套函数组织得多么良好，我总不想将它们全复制粘贴到另一个新函数中。我希望同样的计算函数可以被文本版详单和HTML版详单共用。

要实现复用有许多种方法，而我最喜欢的技术是拆分阶段（154）。这里我的目标是将逻辑分成两部分：一部分计算详单所需的数据，另一部分将数据渲染成文本或HTML。第一阶段会创建一个中转数据结构，再把它传递给第二阶段。

要开始拆分阶段（154），我会先对组成第二阶段的代码应用提炼函数（106）。在这个例子

中，这部分代码就是打印详单的代码，其实也就是statement函数的全部内容。我要把它们与所有嵌套的函数一起抽取到一个新的顶层函数中，并将其命名为renderPlainText。

```
function statement (invoice, plays) {
  return renderPlainText(invoice, plays);
}

function renderPlainText(invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

编译、测试、提交，接着创建一个对象，作为在两个阶段间传递的中转数据结构，然后将它作为第一个参数传递给renderPlainText（然后编译、测试、提交）。

```
function statement (invoice, plays) {
  const statementData = {};
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
```

```
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

现在我要检查一下renderPlainText用到的其他参数。我希望将它们挪到这个中转数据结构里，这样所有计算代码都可以被挪到statement函数中，让renderPlainText只操作通过data参数传进来的数据。

第一步是将顾客（customer）字段添加到中转对象里（编译、测试、提交）。

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

我将performances字段也搬移过去，这样我就可以移除掉renderPlainText的invoice参数（编译、测试、提交）。

顶层作用域...

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances;
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

function renderPlainText...

```
function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += amountFor(perf);
  }
  return result;
}
function totalVolumeCredits() {
  let result = 0;
  for (let perf of data.performances) {
```

```
    result += volumeCreditsFor(perf);
  }
  return result;
}
```

现在，我希望“剧目名称”信息也从中转数据中获得。为此，需要使用`play`中的数据填充`aPerformance`对象（记得编译、测试、提交）。

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances.map(enric
  return renderPlainText(statementData, plays);

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    return result;
}
```

现在我只是简单地返回了一个`aPerformance`对象的副本，但马上我就会往这条记录中添加新的数据。返回副本的原因是，我不想修改传给函数的参数，我总是尽量保持数据不可变（immutable）——可变的状态会很快变成烫手的山芋。

> 在不熟悉 JavaScript 的人看来，`result = Object.assign({}, aPerformance)`的写法可能十分奇怪。它返回的是一个浅副本。虽然我更

希望有个函数来完成此功能，但这个用法已经约定俗成，如果我自己写个函数，在JavaScript程序员看来反而会格格不入。

现在我们已经有了安放`play`字段的地方，可以把数据放进去。我需要对`playFor`和`statement`函数应用搬移函数（198）（然后编译、测试、提交）。

function statement...

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID];
}
```

然后替换`renderPlainText`中对`playFor`的所有引用点，让它们使用新数据（编译、测试、提交）。

function renderPlainText...

```
let result = `Statement for ${data.customer}\n`;
for (let perf of data.performances) {
  result += ` ${perf.play.name}: ${usd(amountFor(perf))} (${p
}
result += `Amount owed is ${usd(totalAmount())}\n`;
result += `You earned ${totalVolumeCredits()} credits\n`;
return result;

function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === aPerformance.play.type) result += Math.flo
  return result;
}

functionamountFor(aPerformance){
  let result = 0;
  switch (aPerformance.play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
    throw new Error(`unknown type: ${aPerformance.play.type}`
  }
  return result;
}
```

接着我使用类似的手法搬移amountFor函数（编译、测试、提交）。

function statement...

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  return result;
}

function amountFor(aPerformance) {...}
```

function renderPlainText...

```
let result = `Statement for ${data.customer}\n`;
for (let perf of data.performances) {
  result += ` ${perf.play.name}: ${usd(perf.amount)} (${perf.
}
result += `Amount owed is ${usd(totalAmount())}\n`;
result += `You earned ${totalVolumeCredits()} credits\n`;
return result;

function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.amount;
  }
  return result;
}
```

接下来搬移观众量积分的计算（编译、测试、提交）。

function statement...

```
function enrichPerformance(aPerformance) {
```

```
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}

function volumeCreditsFor(aPerformance) {...}
```

function renderPlainText...

```
function totalVolumeCredits() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.volumeCredits;
  }
  return result;
}
```

最后，我将两个计算总数的函数搬移到statement函数中。

function statement...

```
const statementData = {};
statementData.customer = invoice.customer;
statementData.performances = invoice.performances.map(enrichP
statementData.totalAmount = totalAmount(statementData);
statementData.totalVolumeCredits = totalVolumeCredits(stateme
return renderPlainText(statementData, plays);

 function totalAmount(data) {...}
```

```
function totalVolumeCredits(data) {...}
```

function renderPlainText...

```
let result = `Statement for ${data.customer}\n`;
for (let perf of data.performances) {
  result += ` ${perf.play.name}: ${usd(perf.amount)} (${perf.
}
result += `Amount owed is ${usd(data.totalAmount)}\n`;
result += `You earned ${data.totalVolumeCredits} credits\n`;
return result;
```

尽管我可以修改函数体，让这些计算总数的函数直接使用statementData变量（反正它在作用域内），但我更喜欢显式地传入函数参数。

等到搬移完成，编译、测试、提交也做完，我便忍不住以管道取代循环（231）对几个地方进行重构。

function renderPlainText...

```
function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}
function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
```

```
}
```

现在我可以把第一阶段的代码提炼到一个独立的函数里了（编译、测试、提交）。

顶层作用域...

```
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays))
}

function createStatementData(invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances.map(enric
  statementData.totalAmount = totalAmount(statementData);
  statementData.totalVolumeCredits = totalVolumeCredits(state
  return statementData;
```

由于两个阶段已经彻底分离，我干脆把它搬移到另一个文件里去（并且修改了返回结果的变量名，与我一贯的编码风格保持一致）。

statement.js...

```
import createStatementData from './createStatementData.js';
```

createStatementData.js...

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerfor
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {...}
    function playFor(aPerformance) {...}
    function amountFor(aPerformance) {...}
    function volumeCreditsFor(aPerformance) {...}
    function totalAmount(data) {...}
    function totalVolumeCredits(data) {...}
```

最后再做一次编译、测试、提交，接下来，要编写一个HTML版本的对账单就很简单了。

statement.js...

```
function htmlStatement (invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}
function renderHtml (data) {
  let result = `<h1>Statement for ${data.customer}</h1>\n`;
  result += "<table>\n";
    result  +=  "<tr><th>play</th><th>seats</th><th>cost</th>
</tr>";
  for (let perf of data.performances) {
          result   +=   `   <tr><td>${perf.play.name}</td>
<td>${perf.audience}</td>`;
    result += `<td>${usd(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
```

```
  result += `<p>Amount owed is <em>${usd(data.totalAmount)}
</em></p>\n`;
  result += `<p>You earned <em>${data.totalVolumeCredits}
</em> credits</p>\n`;
  return result;
}

function usd(aNumber) {...}
```

（我把usd函数也搬移到顶层作用域中，以便renderHtml也能访问它。）

1.7　进展：分离到两个文件（和两个阶段）

现在正是停下来重新回顾一下代码的好时机，思考一下重构的进展。现在我有了两个代码文件。

statement.js

```
import createStatementData from './createStatementData.js';
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays))
}
function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: ${usd(perf.amount)} (${per
  }
  result += `Amount owed is ${usd(data.totalAmount)}\n`;
  result += `You earned ${data.totalVolumeCredits} credits\n`
  return result;
}
function htmlStatement (invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}
function renderHtml (data) {
  let result = `<h1>Statement for ${data.customer}</h1>\n`;
  result += "<table>\n";
    result  +=  "<tr><th>play</th><th>seats</th><th>cost</th>
</tr>";
  for (let perf of data.performances) {
            result   +=   `   <tr><td>${perf.play.name}</td>
<td>${perf.audience}</td>`;
    result += `<td>${usd(perf.amount)}</td></tr>\n`;
  }
```

```
  result += "</table>\n";
   result += `<p>Amount owed is <em>${usd(data.totalAmount)}
</em></p>\n`;
    result += `<p>You earned <em>${data.totalVolumeCredits}
</em> credits</p>\n`;
  return result;
}
function usd(aNumber) {
  return new Intl.NumberFormat("en-US",
                                { style: "currency", currency
                                  minimumFractionDigits: 2 })
}
```

createStatementData.js

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerfor
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }
  function amountFor(aPerformance) {
    let result = 0;
    switch (aPerformance.play.type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
```

```
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
        throw new Error(`unknown type: ${aPerformance.play.ty
    }
    return result;
  }
  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("comedy" === aPerformance.play.type) result += Math.f
    return result;
  }
  function totalAmount(data) {
    return data.performances
      .reduce((total, p) => total + p.amount, 0);
  }
  function totalVolumeCredits(data) {
    return data.performances
      .reduce((total, p) => total + p.volumeCredits, 0);
  }
```

代码行数由我开始重构时的44行增加到了70行（不算`htmlStatement`），这主要是将代码抽取到函数里带来的额外包装成本。虽然代码的行数增加了，但重构也带来了代码可读性的提高。额外的包装将混杂的逻辑分解成可辨别的部分，分离了详单的计算逻辑与样式。这种模块化使我更容易辨别代码的不同部分，了解它们的协作关系。虽说言以简为贵，但可演化的软件却以明确为贵。通过增强代码的模块化，我可以轻易地添加HTML版本的代

码，而无须重复计算部分的逻辑。

> 编程时，需要遵循营地法则：保证你离开时的代码库一定比来时更健康。

其实打印逻辑还可以进一步简化，但当前的代码也够用了。我经常需要在所有可做的重构与添加新特性之间寻找平衡。在当今业界，大多数人面临同样的选择时，似乎多以延缓重构而告终——当然这也是一种选择。我的观点则与营地法则无异：保证离开时的代码库一定比你来时更加健康。完美的境界很难达到，但应该时时都勤加拂拭。

1.8　按类型重组计算过程

接下来我将注意力集中到下一个特性改动：支持更多类型的戏剧，以及支持它们各自的价格计算和观众量积分计算。对于现在的结构，我只需要在计算函数里添加分支逻辑即可。amountFor函数清楚地体现了，戏剧类型在计算分支的选择上起着关键的作用——但这样的分支逻辑很容易随代码堆积而腐坏，除非编程语言提供了更基础的编程语言元素来防止代码堆积。

要为程序引入结构、显式地表达出“计算逻辑的差异是由类型代码确定”有许多途径，不过最自然的解决办法还是使用面向对象世界里的一个经典特性——类型多态。传统的面向对象特性在JavaScript世界一直备受争议，但新的ECMAScript 2015规范有意为类和多态引入了一个相当实用的语法糖。这说明，在合适的场景下使用面向对象是合理的——显然我们这个就是一个合适的使用场景。

我的设想是先建立一个继承体系，它有“喜剧”（comedy）和“悲剧”（tragedy）两个子类，子类各自包含独立的计算逻辑。调用者通过调用一个多态的amount函数，让语言帮你分发到不同的子类

的计算过程中。volumeCredits函数的处理也是如法炮制。为此我需要用到多种重构方法，其中最核心的一招是以多态取代条件表达式（272），将多个同样的类型码分支用多态取代。但在施展以多态取代条件表达式（272）之前，我得先创建一个基本的继承结构。我需要先创建一个类，并将价格计算函数和观众量积分计算函数放进去。

我先从检查计算代码开始。（之前的重构带来的一大好处是，现在我大可以忽略那些格式化代码，只要不改变中转数据结构就行。我可以进一步添加测试来保证中转数据结构不会被意外修改。）

createStatementData.js...

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerfor
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }
  function amountFor(aPerformance) {
```

```
  let result = 0;
  switch (aPerformance.play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${aPerformance.play.ty
  }
  return result;
}
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === aPerformance.play.type) result += Math.f
  return result;
}
function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}
function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
}
```

创建演出计算器

enrichPerformance函数是关键所在，因为正

是它用每场演出的数据来填充中转数据结构。目前它直接调用了计算价格和观众量积分的函数，我需要创建一个类，通过这个类来调用这些函数。由于这个类存放了与每场演出相关数据的计算函数，于是我把它称为演出计算器（performance calculator）。

function createStatementData...

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance);
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

顶层作用域**...**

```
class PerformanceCalculator {
  constructor(aPerformance) {
    this.performance = aPerformance;
  }
}
```

到目前为止，这个新对象还没做什么事。我希望将函数行为搬移进来，这可以从最容易搬移的东

西——play字段开始。严格来讲，我不需要搬移这个字段，因为它并未体现出多态性，但这样可以把所有数据转换集中到一处地方，保证了代码的一致性和清晰度。

为此，我将使用改变函数声明（124）手法将performance的play字段传给计算器。

function createStatementData...

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance,
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

class PerformanceCalculator...

```
class PerformanceCalculator {
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
}
```

（以下行文中我将不再特别提及“编译、测试、提交”循环，我猜你也已经读得有些厌烦了。但我仍会不断重复这个循环。的确，有时我也会厌烦，直到错误又跳出来咬我一下，我才又学会进入小步的节奏。）

将函数搬移进计算器

我要搬移的下一块逻辑，对计算一场演出的价格（amount）来说就尤为重要了。在调整嵌套函数的层级时，我经常将函数挪来挪去，但接下来需要改动到更深入的函数上下文，因此我将小心使用搬移函数（198）来重构它。首先，将`amount`函数的逻辑复制一份到新的上下文中，也就是`PerformanceCalculator`类中。然后微调一下代码，将`aPerformance`改为`this.performance`，将`playFor(aPerformance)`改为`this.play`，使代码适应这个新家。

class PerformanceCalculator...

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
```

```
        result = 40000;
        if (this.performance.audience > 30) {
          result += 1000 * (this.performance.audience - 30);
        }
        break;
      case "comedy":
        result = 30000;
        if (this.performance.audience > 20) {
          result += 10000 + 500 * (this.performance.audience -
        }
        result += 300 * this.performance.audience;
        break;
      default:
        throw new Error(`unknown type: ${this.play.type}`);
    }
    return result;
  }
```

搬移完成后可以编译一下，看看是否有编译错误。我在本地开发环境运行代码时，编译会自动发生，我实际需要做的只是运行一下Babel。编译能帮我发现新函数中潜在的语法错误，语法之外的就帮不上什么忙了。尽管如此，这一步还是很有用。

使新函数适应新家后，我会将原来的函数改造成一个委托函数，让它直接调用新函数。

function createStatementData...

```
function amountFor(aPerformance) {
  return new PerformanceCalculator(aPerformance, playFor(aPer
}
```

现在，我可以执行一次编译、测试、提交，确保代码搬到新家后也能如常工作。之后，我应用内联函数（115），让引用点直接调用新函数（然后编译、测试、提交）。

function createStatementData...

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance,
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

搬移观众量积分计算也遵循同样的流程。

function createStatementData...

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance,
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

class PerformanceCalculator...

```
get volumeCredits() {
  let result = 0;
  result += Math.max(this.performance.audience - 30, 0);
  if ("comedy" === this.play.type) result += Math.floor(this.
  return result;
}
```

使演出计算器表现出多态性

我已将全部计算逻辑搬移到一个类中，是时候将它多态化了。第一步是应用以子类取代类型码（362）引入子类，弃用类型代码。为此，我需要为演出计算器创建子类，并在`createStatementData`中获取对应的子类。要得到正确的子类，我需要将构造函数调用替换为一个普通的函数调用，因为JavaScript的构造函数里无法返回子类。于是我使用以工厂函数取代构造函数（334）。

function createStatementData...

```
function enrichPerformance(aPerformance) {
```

```
  const calculator = createPerformanceCalculator(aPerformance
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

顶层作用域...

```
function createPerformanceCalculator(aPerformance, aPlay) {
  return new PerformanceCalculator(aPerformance, aPlay);
}
```

改造成普通函数后，我就可以在里面创建演出计算器的子类，然后由创建函数决定返回哪一个子类的实例。

顶层作用域...

```
function createPerformanceCalculator(aPerformance, aPlay) {
    switch(aPlay.type) {
    case "tragedy": return new TragedyCalculator(aPerformance
    case "comedy" : return new ComedyCalculator(aPerformance,
    default:
        throw new Error(`unknown type: ${aPlay.type}`);
    }
}

class TragedyCalculator extends PerformanceCalculator {
}
class ComedyCalculator extends PerformanceCalculator {
```

```
}
```

准备好实现多态的类结构后，我就可以继续使用以多态取代条件表达式（272）手法了。

我先从悲剧的价格计算逻辑开始搬移。

class TragedyCalculator...

```
get amount() {
  let result = 40000;
  if (this.performance.audience > 30) {
    result += 1000 * (this.performance.audience - 30);
  }
  return result;
}
```

虽说子类有了这个方法已足以覆盖超类对应的条件分支，但要是你也和我一样偏执，你也许还想在超类的分支上抛一个异常。

class PerformanceCalculator...

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
      throw 'bad thing';
    case "comedy":
```

```
        result = 30000;
        if (this.performance.audience > 20) {
          result += 10000 + 500 * (this.performance.audience -
        }
        result += 300 * this.performance.audience;
        break;
      default:
        throw new Error(`unknown type: ${this.play.type}`);
    }
    return result;
  }
```

虽然我也可以直接删掉处理悲剧的分支，将错误留给默认分支去抛出，但我更喜欢显式地抛出异常——何况这行代码只能再活个几分钟了（这也是我直接抛出一个字符串而不用更好的错误对象的原因）。

再次进行编译、测试、提交。之后，将处理喜剧类型的分支也下移到子类中去。

class ComedyCalculator...

```
  get amount() {
    let result = 30000;
    if (this.performance.audience > 20) {
      result += 10000 + 500 * (this.performance.audience - 20);
    }
    result += 300 * this.performance.audience;
    return result;
  }
```

理论上讲，我可以将超类的amount方法一并移除了，反正它也不应再被调用到。但不删它，给未来的自己留点纪念品也是极好的，顺便可以提醒后来者记得实现这个函数。

class PerformanceCalculator...

```
get amount() {
  throw new Error('subclass responsibility');
}
```

下一个要替换的条件表达式是观众量积分的计算。我回顾了一下前面关于未来戏剧类型的讨论，发现大多数剧类在计算积分时都会检查观众数是否达到30，仅一小部分品类有所不同。因此，将更为通用的逻辑放到超类作为默认条件，出现特殊场景时按需覆盖它，听起来十分合理。于是我将一部分喜剧的逻辑下移到子类。

class PerformanceCalculator...

```
get volumeCredits() {
  return Math.max(this.performance.audience - 30, 0);
}
```

class ComedyCalculator...

```
get volumeCredits() {
  return super.volumeCredits + Math.floor(this.performance.au
}
```

1.9　进展：使用多态计算器来提供数据

又到了观摩代码的时刻，让我们来看看，为计算器引入多态会对代码库有什么影响。

createStatementData.js

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerfor
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const calculator = createPerformanceCalculator(aPerforman
    const result = Object.assign({}, aPerformance);
    result.play = calculator.play;
    result.amount = calculator.amount;
    result.volumeCredits = calculator.volumeCredits;
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }
  function totalAmount(data) {
    return data.performances
      .reduce((total, p) => total + p.amount, 0);
  }
  function totalVolumeCredits(data) {
```

```
    return data.performances
      .reduce((total, p) => total + p.volumeCredits, 0);
  }
}
function createPerformanceCalculator(aPerformance, aPlay) {
    switch(aPlay.type) {
    case "tragedy": return new TragedyCalculator(aPerformance
    case "comedy" : return new ComedyCalculator(aPerformance,
    default:
        throw new Error(`unknown type: ${aPlay.type}`);
    }
}
class PerformanceCalculator {
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
  get amount() {
    throw new Error('subclass responsibility');
  }
  get volumeCredits() {
    return Math.max(this.performance.audience - 30, 0);
  }
}
class TragedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 40000;
    if (this.performance.audience > 30) {
      result += 1000 * (this.performance.audience - 30);
    }
    return result;
  }
}
class ComedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 30000;
    if (this.performance.audience > 20) {
      result += 10000 + 500 * (this.performance.audience - 20
    }
    result += 300 * this.performance.audience;
    return result;
  }
  get volumeCredits() {
    return super.volumeCredits + Math.floor(this.performance.
  }
}
```

代码量仍然有所增加，因为我再次整理了代码结构。新结构带来的好处是，不同戏剧种类的计算各自集中到了一处地方。如果大多数修改都涉及特定类型的计算，像这样按类型进行分离就很有意义。当添加新剧种时，只需要添加一个子类，并在创建函数中返回它。

这个示例还揭示了一些关于此类继承方案何时适用的洞见。上面我将条件分支的查找从两个不同的函数（amountFor和volumeCreditsFor）搬移到一个集中的构造函数createPerformanceCalculator中。有越多的函数依赖于同一套类型进行多态，这种继承方案就越有益处。

除了这样设计，还有另一种可能的方案，那就是让createStatementData返回计算器实例本身，而非自己拿到计算器来填充中转数据结构。JavaScript的类设计有不少好特性，例如，取值函数用起来就像普通的数据存取。我在考量是“直接返回实例本身”还是“返回计算好的中转数据”时，主要看数据的使用者是谁。在这个例子中，我更想通过中转数据结构来展示如何以此隐藏计算器背后的多态设计。

1.10　结语

这是一个简单的例子，但我希望它能让你对“重构怎么做”有一点感觉。例中我已经示范了数种重构手法，包括提炼函数（106）、内联变量（123）、搬移函数（198）和以多态取代条件表达式（272）等。

本章的重构有3个较为重要的节点，分别是：将原函数分解成一组嵌套的函数、应用拆分阶段（154）分离计算逻辑与输出格式化逻辑，以及为计算器引入多态性来处理计算逻辑。每一步都给代码添加了更多的结构，以便我能更好地表达代码的意图。

一般来说，重构早期的主要动力是尝试理解代码如何工作。通常你需要先通读代码，找到一些感觉，然后再通过重构将这些感觉从脑海里搬回到代码中。清晰的代码更容易理解，使你能够发现更深层次的设计问题，从而形成积极正向的反馈环。当然，这个示例仍有值得改进的地方，但现在测试仍能全部通过，代码相比初见时已经有了巨大的改善，所以我已经可以满足了。

我谈论的是如何改善代码，但什么样的代码才

算好代码，程序员们有很多争论。我偏爱小的、命名良好的函数，也知道有些人反对这个观点。如果我们说这只关乎美学，只是各花入各眼，没有好坏高低之分，那除了诉诸个人品味，就没有任何客观事实依据了。但我坚信，这不仅关乎个人品味，而且是有客观标准的。我认为，好代码的检验标准就是人们是否能轻而易举地修改它。好代码应该直截了当：有人需要修改代码时，他们应能轻易找到修改点，应该能快速做出更改，而不易引入其他错误。一个健康的代码库能够最大限度地提升我们的生产力，支持我们更快、更低成本地为用户添加新特性。为了保持代码库的健康，就需要时刻留意现状与理想之间的差距，然后通过重构不断接近这个理想。

好代码的检验标准就是人们是否能轻而易举地修改它。

这个示例告诉我们最重要的一点就是重构的节奏感。无论何时，当我向人们展示我如何重构时，无人不讶异于我的步子之小，并且每一步都保证代码处于编译通过和测试通过的可工作状态。20年前，当Kent Beck在底特律的一家宾馆里向我展示同样的手法时，我也报以同样的震撼。开展高效有序

的重构，关键的心得是：小的步子可以更快前进，请保持代码永远处于可工作状态，小步修改累积起来也能大大改善系统的设计。这几点请君牢记，其余的我已无需多言。

第2章　重构的原则

前一章所举的例子应该已经让你对重构有了一个良好的感觉。现在，我们应该回头看看重构的一些大原则。

2.1　何谓重构

一线的实践者们经常很随意地使用“重构”这个词——软件开发领域的很多词汇都有此待遇。我使用这个词的方式比较严谨，并且我发现这种严谨的方式很有好处。（下列定义与本书第1版中给出的定义一样。）“重构”这个词既可以用作名词也可以用作动词。名词形式的定义是：

重构（名词）：对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。

这个定义适用于我在前面的例子中提到的那些有名字的重构，例如提炼函数（106）和以多态取代条件表达式（272）。

动词形式的定义是：

重构（动词）：使用一系列重构手法，在不改变软件可观察行为的前提下，调整其结构。

所以，我可能会花一两个小时进行重构（动词），其间我会使用几十个不同的重构（名词）。

过去十几年，这个行业里的很多人用“重构”这

个词来指代任何形式的代码清理，但上面的定义所指的是一种特定的清理代码的方式。重构的关键在于运用大量微小且保持软件行为的步骤，一步步达成大规模的修改。每个单独的重构要么很小，要么由若干小步骤组合而成。因此，在重构的过程中，我的代码很少进入不可工作的状态，即便重构没有完成，我也可以在任何时刻停下来。

> 如果有人说他们的代码在重构过程中有一两天时间不可用，基本上可以确定，他们在做的事不是重构。

我会用“结构调整”（restructuring）来泛指对代码库进行的各种形式的重新组织或清理，重构则是特定的一类结构调整。刚接触重构的人看我用很多小步骤完成似乎可以一大步就能做完的事，可能会觉得这样很低效。但小步前进能让我走得更快，因为这些小步骤能完美地彼此组合，而且——更关键的是——整个过程中我不会花任何时间来调试。

在上述定义中，我用了“可观察行为”的说法。它的意思是，整体而言，经过重构之后的代码所做的事应该与重构之前大致一样。这个说法并非完全严格，并且我是故意保留这点儿空间的：重构之后

的代码不一定与重构前行为完全一致。比如说，提炼函数（106）会改变函数调用栈，因此程序的性能就会有所改变；改变函数声明（124）和搬移函数（198）等重构经常会改变模块的接口。不过就用户应该关心的行为而言，不应该有任何改变。如果我在重构过程中发现了任何bug，重构完成后同样的bug应该仍然存在（不过，如果潜在的bug还没有被任何人发现，也可以当即把它改掉）。

重构与性能优化有很多相似之处：两者都需要修改代码，并且两者都不会改变程序的整体功能。两者的差别在于其目的：重构是为了让代码“更容易理解，更易于修改”。这可能使程序运行得更快，也可能使程序运行得更慢。在性能优化时，我只关心让程序运行得更快，最终得到的代码有可能更难理解和维护，对此我有心理准备。

2.2　两顶帽子

Kent Beck提出了“两顶帽子”的比喻。使用重构技术开发软件时，我把自己的时间分配给两种截然不同的行为：添加新功能和重构。添加新功能时，我不应该修改既有代码，只管添加新功能。通过添加测试并让测试正常运行，我可以衡量自己的工作进度。重构时我就不能再添加功能，只管调整代码的结构。此时我不应该添加任何测试（除非发现有先前遗漏的东西），只在绝对必要（用以处理接口变化）时才修改测试。

软件开发过程中，我可能会发现自己经常变换帽子。首先我会尝试添加新功能，然后会意识到：如果把程序结构改一下，功能的添加会容易得多。于是我换一顶帽子，做一会儿重构工作。程序结构调整好后，我又换上原先的帽子，继续添加新功能。新功能正常工作后，我又发现自己的编码造成程序难以理解，于是又换上重构帽子……整个过程或许只花10分钟，但无论何时我都清楚自己戴的是哪一顶帽子，并且明白不同的帽子对编程状态提出的不同要求。

2.3　为何重构

我不想把重构说成是包治百病的万灵丹，它绝对不是所谓的“银弹”。不过它的确很有价值，尽管它不是一颗“银弹”，却可以算是一把“银钳子”，可以帮你始终良好地控制自己的代码。重构是一个工具，它可以（并且应该）用于以下几个目的。

重构改进软件的设计

如果没有重构，程序的内部设计（或者叫架构）会逐渐腐败变质。当人们只为短期目的而修改代码时，他们经常没有完全理解架构的整体设计，于是代码逐渐失去了自己的结构。程序员越来越难通过阅读源码来理解原来的设计。代码结构的流失有累积效应。越难看出代码所代表的设计意图，就越难保护其设计，于是设计就腐败得越快。经常性的重构有助于代码维持自己该有的形态。

完成同样一件事，设计欠佳的程序往往需要更多代码，这常常是因为代码在不同的地方使用完全相同的语句做同样的事，因此改进设计的一个重要

方向就是消除重复代码。代码量减少并不会使系统运行更快，因为这对程序的资源占用几乎没有任何明显影响。然而代码量减少将使未来可能的程序修改动作容易得多。代码越多，做正确的修改就越困难，因为有更多代码需要理解。我在这里做了点儿修改，系统却不如预期那样工作，因为我没有修改另一处——那里的代码做着几乎完全一样的事情，只是所处环境略有不同。消除重复代码，我就可以确定所有事物和行为在代码中只表述一次，这正是优秀设计的根本。

重构使软件更容易理解

所谓程序设计，很大程度上就是与计算机对话：我编写代码告诉计算机做什么事，而它的响应是按照我的指示精确行动。一言以蔽之，我所做的就是填补“我想要它做什么”和“我告诉它做什么”之间的缝隙。编程的核心就在于“准确说出我想要的”。然而别忘了，除了计算机外，源码还有其他读者：几个月之后可能会有另一位程序员尝试读懂我的代码并对其做一些修改。我们很容易忘记这这位读者，但他才是最重要的。计算机是否多花了几个时钟周期来编译，又有什么关系呢？如果一个程序员花费一周时间来修改某段代码，那才要命呢

——如果他理解了我的代码，这个修改原本只需一小时。

问题在于，当我努力让程序运转的时候，我不会想到未来出现的那个开发者。是的，我们应该改变一下开发节奏，让代码变得更易于理解。重构可以帮我让代码更易读。开始进行重构前，代码可以正常运行，但结构不够理想。在重构上花一点点时间，就可以让代码更好地表达自己的意图——更清晰地说出我想要做的。

关于这一点，我没必要表现得多么无私。很多时候那个未来的开发者就是我自己。此时重构就显得尤其重要了。我是一个很懒惰的程序员，我的懒惰表现形式之一就是：总是记不住自己写过的代码。事实上，对于任何能够立刻查阅的东西，我都故意不去记它，因为我怕把自己的脑袋塞爆。我总是尽量把该记住的东西写进代码里，这样我就不必记住它了。这么一来，下班后我还可以喝上两杯Maudite啤酒，不必太担心它杀光我的脑细胞。

重构帮助找到bug

对代码的理解，可以帮我找到bug。我承认我不太擅长找bug。有些人只要盯着一大段代码就可

以找出里面的bug，我不行。但我发现，如果对代码进行重构，我就可以深入理解代码的所作所为，并立即把新的理解反映在代码当中。搞清楚程序结构的同时，我也验证了自己所做的一些假设，于是想不把bug揪出来都难。

这让我想起了Kent Beck经常形容自己的一句话："我不是一个特别好的程序员，我只是一个有着一些特别好的习惯的还不错的程序员。"重构能够帮助我更有效地写出健壮的代码。

重构提高编程速度

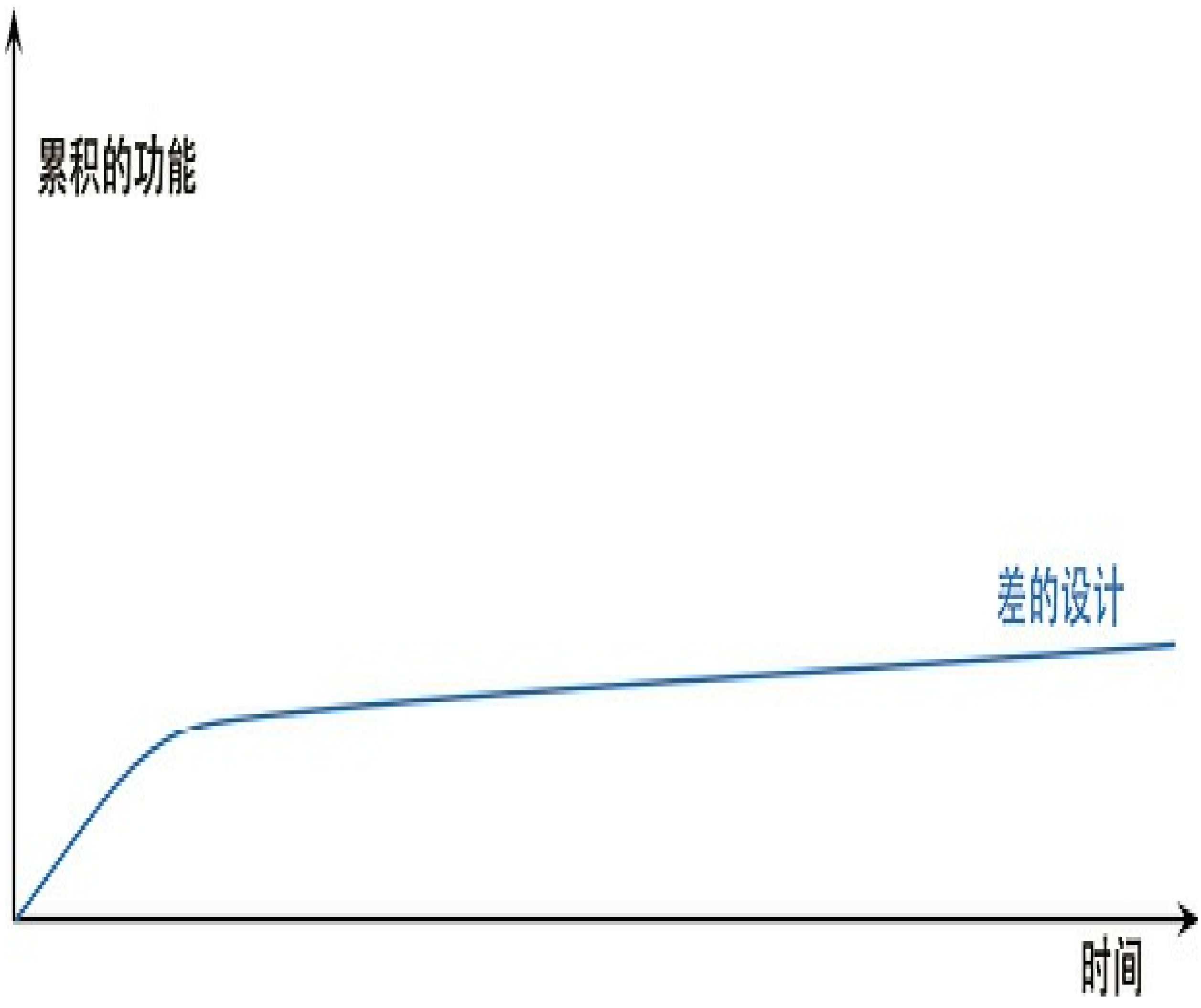
最后，前面的一切都归结到了这一点：重构帮我更快速地开发程序。

听起来有点儿违反直觉。当我谈到重构时，人们很容易看出它能够提高质量。改善设计、提升可读性、减少bug，这些都能提高质量。但花在重构上的时间，难道不是在降低开发速度吗？

当我跟那些在一个系统上工作较长时间的软件开发者交谈时，经常会听到这样的故事：一开始他们进展很快，但如今想要添加一个新功能需要的时间就要长得多。他们需要花越来越多的时间去考虑

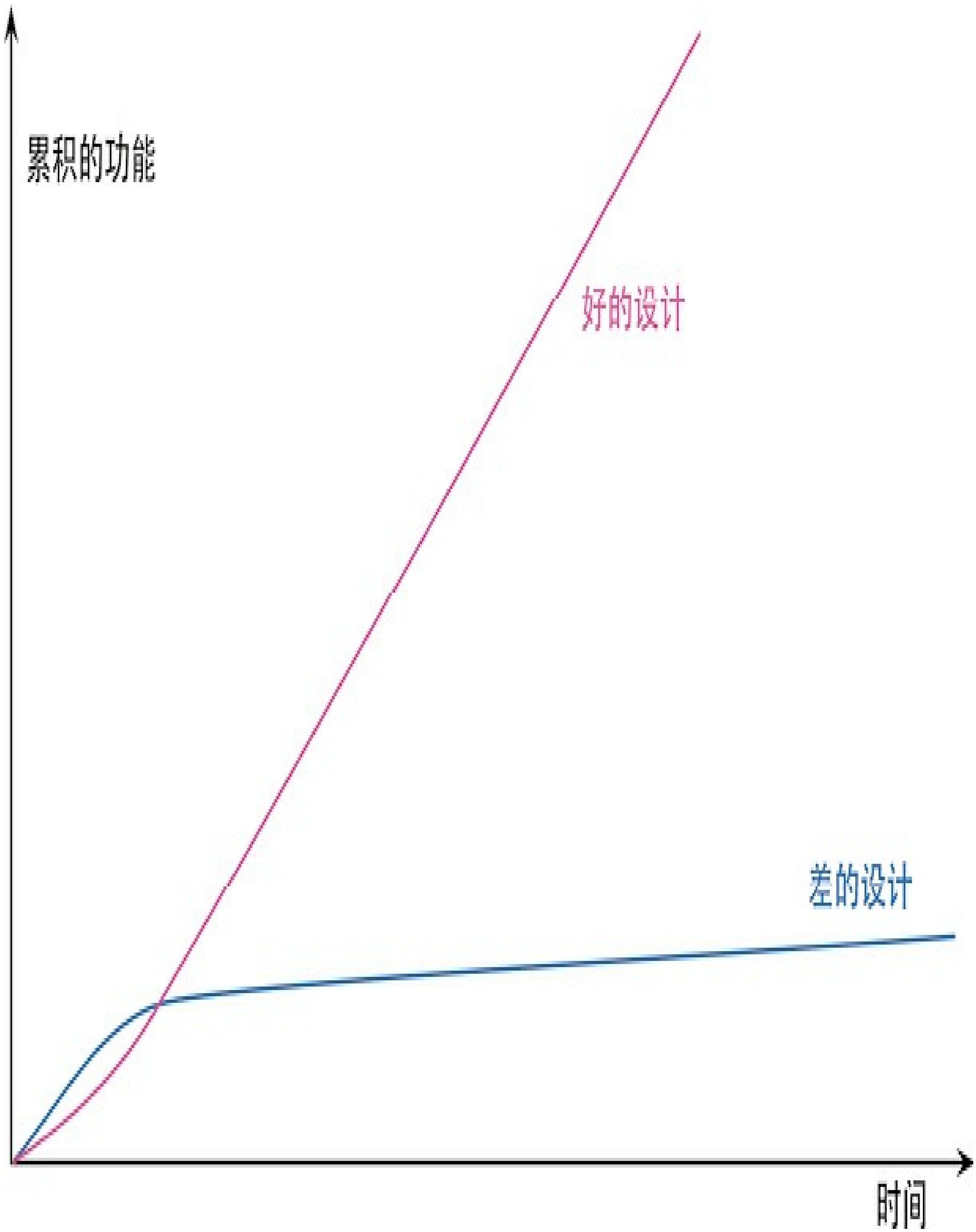
如何把新功能塞进现有的代码库，不断蹦出来的bug修复起来也越来越慢。代码库看起来就像补丁摞补丁，需要细致的考古工作才能弄明白整个系统是如何工作的。这份负担不断拖慢新增功能的速度，到最后程序员恨不得从头开始重写整个系统。

下面这幅图可以描绘他们经历的困境。



但有些团队的境遇则截然不同。他们添加新功能的速度越来越快，因为他们能利用已有的功能，

基于已有的功能快速构建新功能。



两种团队的区别就在于软件的内部质量。需要添加新功能时，内部质量良好的软件让我可以很容易找到在哪里修改、如何修改。良好的模块划分使我只需要理解代码库的一小部分，就可以做出修改。如果代码很清晰，我引入bug的可能性就会变小，即使引入了bug，调试也会容易得多。理想情况下，我的代码库会逐步演化成一个平台，在其上可以很容易地构造与其领域相关的新功能。

我把这种现象称为“设计耐久性假说”：通过投入精力改善内部设计，我们增加了软件的耐久性，从而可以更长时间地保持开发的快速。我还无法科学地证明这个理论，所以我说它是一个“假说”。但我的经验，以及我在职业生涯中认识的上百名优秀程序员的经验，都支持这个假说。

20年前，行业的陈规认为：良好的设计必须在开始编程之前完成，因为一旦开始编写代码，设计就只会逐渐腐败。重构改变了这个图景。现在我们可以改善已有代码的设计，因此我们可以先做一个设计，然后不断改善它，哪怕程序本身的功能也在不断发生着变化。由于预先做出良好的设计非常困难，想要既体面又快速地开发功能，重构必不可少。

2.4 何时重构

在我编程的每个小时，我都会做重构。有几种方式可以把重构融入我的工作过程里。

三次法则

> Don Roberts给了我一条准则：第一次做某件事时只管去做；第二次做类似的事会产生反感，但无论如何还是可以去做；第三次再做类似的事，你就应该重构。
>
> 正如老话说的：事不过三，三则重构。

预备性重构：让添加新功能更容易

重构的最佳时机就在添加新功能之前。在动手添加新功能之前，我会看看现有的代码库，此时经常会发现：如果对代码结构做一点微调，我的工作会容易得多。也许已经有个函数提供了我需要的大部分功能，但有几个字面量的值与我的需要略有冲突。如果不做重构，我可能会把整个函数复制过来，修改这几个值，但这就会导致重复代码——如果将来我需要做修改，就必须同时修改两处（更麻

烦的是，我得先找到这两处）。而且，如果将来我还需要一个类似又略有不同的功能，就只能再复制粘贴一次，这可不是个好主意。所以我戴上重构的帽子，使用函数参数化（310）。做完这件事以后，接下来我就只需要调用这个函数，传入我需要的参数。

> 这就好像我要往东去100公里。我不会往东一头把车开进树林，而是先往北开20公里上高速，然后再向东开100公里。后者的速度比前者要快上3倍。如果有人催着你“赶快直接去那儿”，有时你需要说：“等等，我要先看看地图，找出最快的路径。”这就是预备性重构于我的意义。
>
> ——Jessica Kerr

修复bug时的情况也是一样。在寻找问题根因时，我可能会发现：如果把3段一模一样且都会导致错误的代码合并到一处，问题修复起来会容易得多。或者，如果把某些更新数据的逻辑与查询逻辑分开，会更容易避免造成错误的逻辑纠缠。用重构改善这些情况，在同样场合再次出现同样bug的概率也会降低。

帮助理解的重构：使代码更易懂

我需要先理解代码在做什么，然后才能着手修改。这段代码可能是我写的，也可能是别人写的。一旦我需要思考“这段代码到底在做什么”，我就会自问：能不能重构这段代码，令其一目了然？我可能看见了一段结构糟糕的条件逻辑，也可能希望复用一个函数，但花费了几分钟才弄懂它到底在做什么，因为它的函数命名实在是太糟糕了。这些都是重构的机会。

看代码时，我会在脑海里形成一些理解，但我的记性不好，记不住那么多细节。正如Ward Cunningham所说，通过重构，我就把脑子里的理解转移到了代码本身。随后我运行这个软件，看它是否正常工作，来检查这些理解是否正确。如果把对代码的理解植入代码中，这份知识会保存得更久，并且我的同事也能看到。

重构带来的帮助不仅发生在将来——常常是立竿见影。我会先在一些小细节上使用重构来帮助理解，给一两个变量改名，让它们更清楚地表达意图，以方便理解，或是将一个长函数拆成几个小函数。当代码变得更清晰一些时，我就会看见之前看不见的设计问题。如果不做前面的重构，我可能永

远都看不见这些设计问题，因为我不够聪明，无法在脑海中推演所有这些变化。Ralph Johnson说，这些初步的重构就像扫去窗上的尘埃，使我们得以看到窗外的风景。在研读代码时，重构会引领我获得更高层面的理解，如果只是阅读代码很难有此领悟。有些人以为这些重构只是毫无意义地把玩代码，他们没有意识到，缺少了这些细微的整理，他们就无法看到隐藏在一片混乱背后的机遇。

捡垃圾式重构

帮助理解的重构还有一个变体：我已经理解代码在做什么，但发现它做得不好，例如逻辑不必要地迂回复杂，或者两个函数几乎完全相同，可以用一个参数化的函数取而代之。这里有一个取舍：我不想从眼下正要完成的任务上跑题太多，但我也不想把垃圾留在原地，给将来的修改增加麻烦。如果我发现的垃圾很容易重构，我会马上重构它；如果重构需要花一些精力，我可能会拿一张便笺纸把它记下来，完成当下的任务再回来重构它。

当然，有时这样的垃圾需要好几个小时才能解决，而我又有更紧急的事要完成。不过即便如此，稍微花一点工夫做一点儿清理，通常都是值得的。

正如野营者的老话所说：至少要让营地比你到达时更干净。如果每次经过这段代码时都把它变好一点点，积少成多，垃圾总会被处理干净。重构的妙处就在于，每个小步骤都不会破坏代码——所以，有时一块垃圾在好几个月之后才终于清理干净，但即便每次清理并不完整，代码也不会被破坏。

有计划的重构和见机行事的重构

上面的例子——预备性重构、帮助理解的重构、捡垃圾式重构——都是见机行事的：我并不专门安排一段时间来重构，而是在添加功能或修复bug的同时顺便重构。这是我自然的编程流的一部分。不管是要添加功能还是修复bug，重构对我当下的任务有帮助，而且让我未来的工作更轻松。这是一件很重要而又常被误解的事：重构不是与编程割裂的行为。你不会专门安排时间重构，正如你不会专门安排时间写`if`语句。我的项目计划上没有专门留给重构的时间，绝大多数重构都在我做其他事的过程中自然发生。

肮脏的代码必须重构，但漂亮的代码

也需要很多重构。

还有一种常见的误解认为，重构就是人们弥补过去的错误或者清理肮脏的代码。当然，如果遇上了肮脏的代码，你必须重构，但漂亮的代码也需要很多重构。在写代码时，我会做出很多权衡取舍：参数化需要做到什么程度？函数之间的边界应该划在哪里？对于昨天的功能完全合理的权衡，在今天要添加新功能时可能就不再合理。好在，当我需要改变这些权衡以反映现实情况的变化时，整洁的代码重构起来会更容易。

> 每次要修改时，首先令修改很容易（警告：这件事有时会很难），然后再进行这次容易的修改。
>
> ——Kent Beck

长久以来，人们认为编写软件是一个累加的过程：要添加新功能，我们就应该增加新代码。但优秀的程序员知道，添加新功能最快的方法往往是先修改现有的代码，使新功能容易被加入。所以，软件永远不应该被视为“完成”。每当需要新能力时，软件就应该做出相应的改变。越是在已有代码中，

这样的改变就越显重要。

不过，说了这么多，并不表示有计划的重构总是错的。如果团队过去忽视了重构，那么常常会需要专门花一些时间来优化代码库，以便更容易添加新功能。在重构上花一个星期的时间，会在未来几个月里发挥价值。有时，即便团队做了日常的重构，还是会有问题在某个区域逐渐累积长大，最终需要专门花些时间来解决。但这种有计划的重构应该很少，大部分重构应该是不起眼的、见机行事的。

我听过的一条建议是：将重构与添加新功能在版本控制的提交中分开。这样做的一大好处是可以各自独立地审阅和批准这些提交。但我并不认同这种做法。重构常常与新添功能紧密交织，不值得花工夫把它们分开。并且这样做也使重构脱离了上下文，使人看不出这些“重构提交”的价值。每个团队应该尝试并找出适合自己的工作方式，只是要记住：分离重构提交并不是毋庸置疑的原则，只有当你真的感到有益时，才值得这样做。

长期重构

大多数重构可以在几分钟——最多几小时——

内完成。但有一些大型的重构可能要花上几个星期，例如要替换一个正在使用的库，或者将整块代码抽取到一个组件中并共享给另一支团队使用，再或者要处理一大堆混乱的依赖关系，等等。

即便在这样的情况下，我仍然不愿让一支团队专门做重构。可以让整个团队达成共识，在未来几周时间里逐步解决这个问题，这经常是一个有效的策略。每当有人靠近“重构区”的代码，就把它朝想要改进的方向推动一点。这个策略的好处在于，重构不会破坏代码——每次小改动之后，整个系统仍然照常工作。例如，如果想替换掉一个正在使用的库，可以先引入一层新的抽象，使其兼容新旧两个库的接口。一旦调用方已经完全改为使用这层抽象，替换下面的库就会容易得多。（这个策略叫作Branch By Abstraction[mf-bba]。）

复审代码时重构

一些公司会做常规的代码复审（code review），因为这种活动可以改善开发状况。代码复审有助于在开发团队中传播知识，也有助于让较有经验的开发者把知识传递给比较欠缺经验的人，并帮助更多人理解大型软件系统中的更多部分。代

码复审对于编写清晰代码也很重要。我的代码也许对我自己来说很清晰，对他人则不然。这是无法避免的，因为要让开发者设身处地为那些不熟悉自己所作所为的人着想，实在太困难了。代码复审也让更多人有机会提出有用的建议，毕竟我在一个星期之内能够想出的好点子很有限。如果能得到别人的帮助，我的生活会滋润得多，所以我总是期待更多复审。

　　我发现，重构可以帮助我复审别人的代码。开始重构前我可以先阅读代码，得到一定程度的理解，并提出一些建议。一旦想到一些点子，我就会考虑是否可以通过重构立即轻松地实现它们。如果可以，我就会动手。这样做了几次以后，我可以更清楚地看到，当我的建议被实施以后，代码会是什么样。我不必想象代码应该是什么样，我可以真实看见。于是我可以获得更高层次的认识。如果不进行重构，我永远无法得到这样的认识。

　　重构还可以帮助代码复审工作得到更具体的结果。不仅获得建议，而且其中许多建议能够立刻实现。最终你将从实践中得到比以往多得多的成就感。

　　至于如何在代码复审的过程中加入重构，这要取决于复审的形式。在常见的pull request模式下，复审者独自浏览代码，代码的作者不在旁边，此时

进行重构效果并不好。如果代码的原作者在旁边会好很多，因为作者能提供关于代码的上下文信息，并且充分认同复审者进行修改的意图。对我个人而言，与原作者肩并肩坐在一起，一边浏览代码一边重构，体验是最佳的。这种工作方式很自然地导向结对编程：在编程的过程中持续不断地进行代码复审。

怎么对经理说

“该怎么跟经理说重构的事?”这是我最常被问到的一个问题。毋庸讳言，我见过一些场合，“重构”被视为一个脏词——经理（和客户）认为重构要么是在弥补过去犯下的错误，要么是不增加价值的无用功。如果团队又计划了几周时间专门做重构，情况就更糟糕了——如果他们做的其实还不是重构，而是不加小心的结构调整，然后又对代码库造成了破坏，那可就真是糟透了。

如果这位经理懂技术，能理解“设计耐久性假说”，那么向他说明重构的意义应该不会很困难。这样的经理应该会鼓励日常的重构，并主动寻找团队日常重构做得不够的征兆。虽然“团队做了太多重构”的情况确实也发生过，但比起做得不够的情

况要罕见得多了。

当然，很多经理和客户不具备这样的技术意识，他们不理解代码库的健康对生产率的影响。这种情况下我会给团队一个较有争议的建议：不要告诉经理！

这是在搞破坏吗？我不这样想。软件开发者都是专业人士。我们的工作就是尽可能快速创造出高效软件。我的经验告诉我，对于快速创造软件，重构可带来巨大帮助。如果需要添加新功能，而原本设计却又使我无法方便地修改，我发现先重构再添加新功能会更快些。如果要修补错误，就得先理解软件的工作方式，而我发现重构是理解软件的最快方式。受进度驱动的经理要我尽可能快速完成任务，至于怎么完成，那就是我的事了。我领这份工资，是因为我擅长快速实现新功能；我认为最快的方式就是重构，所以我就重构。

何时不应该重构

听起来好像我一直在提倡重构，但确实有一些不值得重构的情况。

如果我看见一块凌乱的代码，但并不需要修改

它，那么我就不需要重构它。如果丑陋的代码能被隐藏在一个API之下，我就可以容忍它继续保持丑陋。只有当我需要理解其工作原理时，对其进行重构才有价值。

另一种情况是，如果重写比重构还容易，就别重构了。这是个困难的决定。如果不花一点儿时间尝试，往往很难真实了解重构一块代码的难度。决定到底应该重构还是重写，需要良好的判断力与丰富的经验，我无法给出一条简单的建议。

2.5　重构的挑战

每当有人大力推荐一种技术、工具或者架构时，我总是会观察这东西会遇到哪些挑战，毕竟生活中很少有晴空万里的好事。你需要了解一件事背后的权衡取舍，才能决定何时何地应用它。我认为重构是一种很有价值的技术，大多数团队都应该更多地重构，但它也不是完全没有挑战的。有必要充分了解重构会遇到的挑战，这样才能做出有效应对。

延缓新功能开发

如果你读了前面一小节，我对这个挑战的回应便已经很清楚了。尽管重构的目的是加快开发速度，但是，仍旧很多人认为，花在重构的时间是在拖慢新功能的开发进度。“重构会拖慢进度”这种看法仍然很普遍，这可能是导致人们没有充分重构的最大阻力所在。

☼ 重构的唯一目的就是让我们开发更快，用更少的工作量创造更大的价值。

有一种情况确实需要权衡取舍。我有时会看到一个（大规模的）重构很有必要进行，而马上要添加的功能非常小，这时我会更愿意先把新功能加上，然后再做这次大规模重构。做这个决定需要判断力——这是我作为程序员的专业能力之一。我很难描述决定的过程，更无法量化决定的依据。

我清楚地知道，预备性重构常会使修改更容易，所以如果做一点儿重构能让新功能实现更容易，我一定会做。如果一个问题我已经见过，此时我也会更倾向于重构它——有时我就得先看见一块丑陋的代码几次，然后才能提起劲头来重构它。也就是说，如果一块代码我很少触碰，它不会经常给我带来麻烦，那么我就倾向于不去重构它。如果我还没想清楚究竟应该如何优化代码，那么我可能会延迟重构；当然，有的时候，即便没想清楚优化的方向，我也会先做些实验，试试看能否有所改进。

我从同事那里听到的证据表明，在我们这个行业里，重构不足的情况远多于重构过度的情况。换句话说，绝大多数人应该尝试多做重构。代码库的健康与否，到底会对生产率造成多大的影响，很多人可能说不出来，因为他们没有太多在健康的代码

库上工作的经历——轻松地把现有代码组合配置，快速构造出复杂的新功能，这种强大的开发方式他们没有体验过。

虽然我们经常批评管理者以“保障开发速度”的名义压制重构，其实程序员自己也经常这么干。有时他们自己觉得不应该重构，其实他们的领导还挺希望他们做一些重构的。如果你是一支团队的技术领导，一定要向团队成员表明，你重视改善代码库健康的价值。合理判断何时应该重构、何时应该暂时不重构，这样的判断力需要多年经验积累。对于重构缺乏经验的年轻人需要有意的指导，才能帮助他们加速经验积累的过程。

有些人试图用“整洁的代码”“良好的工程实践”之类道德理由来论证重构的必要性，我认为这是个陷阱。重构的意义不在于把代码库打磨得闪闪发光，而是纯粹经济角度出发的考量。我们之所以重构，因为它能让我们更快——添加功能更快，修复bug更快。一定要随时记住这一点，与别人交流时也要不断强调这一点。重构应该总是由经济利益驱动。程序员、经理和客户越理解这一点，“好的设计”那条曲线就会越经常出现。

代码所有权

很多重构手法不仅会影响一个模块内部，还会影响该模块与系统其他部分的关系。比如我想给一个函数改名，并且我也能找到该函数的所有调用者，那么我只需运用改变函数声明（124），在一次重构中修改函数声明和调用者。但即便这么简单的一个重构，有时也无法实施：调用方代码可能由另一支团队拥有，而我没有权限写入他们的代码库；这个函数可能是一个提供给客户的API，这时我根本无法知道是否有人使用它，至于谁在用、用得有多频繁就更是一无所知。这样的函数属于已发布接口（published interface）：接口的使用者（客户端）与声明者彼此独立，声明者无权修改使用者的代码。

代码所有权的边界会妨碍重构，因为一旦我自作主张地修改，就一定会破坏使用者的程序。这不会完全阻止重构，我仍然可以做很多重构，但确实会对重构造成约束。为了给一个函数改名，我需要使用函数改名（124），但同时也得保留原来的函数声明，使其把调用传递给新的函数。这会让接口变复杂，但这就是为了避免破坏使用者的系统而不得不付出的代价。我可以把旧的接口标记为“不推荐使用”（deprecated），等一段时间之后最终让其退休；但有些时候，旧的接口必须一直保留下去。

由于这些复杂性，我建议不要搞细粒度的强代

码所有制。有些组织喜欢给每段代码都指定唯一的所有者，只有这个人能修改这段代码。我曾经见过一支只有三个人的团队以这种方式运作，每个程序员都要给另外两人发布接口，随之而来的就是接口维护的种种麻烦。如果这三个人都直接去代码库里做修改，事情会简单得多。我推荐团队代码所有制，这样一支团队里的成员都可以修改这个团队拥有的代码，即便最初写代码的是别人。程序员可能各自分工负责系统的不同区域，但这种责任应该体现为监控自己责任区内发生的修改，而不是简单粗暴地禁止别人修改。

这种较为宽容的代码所有制甚至可以应用于跨团队的场合。有些团队鼓励类似于开源的模型：B团队的成员也可以在一个分支上修改A团队的代码，然后把提交发送给A团队去审核。这样一来，如果团队想修改自己的函数，他们就可以同时修改该函数的客户端的代码；只要客户端接受了他们的修改，就可以删掉旧的函数声明了。对于涉及多个团队的大系统开发，在“强代码所有制”和“混乱修改”两个极端之间，这种类似开源的模式常常是一个合适的折中。

分支

很多团队采用这样的版本控制实践：每个团队成员各自在代码库的一条分支上工作，进行相当大量的开发之后，才把各自的修改合并回主线分支（这条分支通常叫master或trunk），从而与整个团队分享。常见的做法是在分支上开发完整的功能，直到功能可以发布到生产环境，才把该分支合并回主线。这种做法的拥趸声称，这样能保持主线不受尚未完成的代码侵扰，能保留清晰的功能添加的版本记录，并且在某个功能出问题时能容易地撤销修改。

这样的特性分支有其缺点。在隔离的分支上工作得越久，将完成的工作集成（integrate）回主线就会越困难。为了减轻集成的痛苦，大多数人的办法是频繁地从主线合并（merge）或者变基（rebase）到分支。但如果有几个人同时在各自的特性分支上工作，这个办法并不能真正解决问题，因为合并与集成是两回事。如果我从主线合并到我的分支，这只是一个单向的代码移动——我的分支发生了修改，但主线并没有。而“集成”是一个双向的过程：不仅要把主线的修改拉（pull）到我的分支上，而且要把我这里修改的结果推（push）回到主线上，两边都会发生修改。假如另一名程序员Rachel正在她的分支上开发，我是看不见她的修改的，直到她将自己的修改与主线集成；此时我就必须把她的修改合并到我的特性分支，这可能需要相

当的工作量。其中困难的部分是处理语义变化。现代版本控制系统都能很好地合并程序文本的复杂修改，但对于代码的语义它们一无所知。如果我修改了一个函数的名字，版本控制工具可以很轻松地将我的修改与Rachel的代码集成。但如果在集成之前，她在自己的分支里新添调用了这个被我改名的函数，集成之后的代码就会被破坏。

分支合并本来就是一个复杂的问题，随着特性分支存在的时间加长，合并的难度会指数上升。集成一个已经存在了4个星期的分支，较之集成存在了2个星期的分支，难度可不止翻倍。所以很多人认为，应该尽量缩短特性分支的生存周期，比如只有一两天。还有一些人（比如我本人）认为特性分支的生命还应该更短，我们采用的方法叫作持续集成（Continuous Integration，CI），也叫“基于主干开发”（Trunk-Based Development）。在使用CI时，每个团队成员每天至少向主线集成一次。这个实践避免了任何分支彼此差异太大，从而极大地降低了合并的难度。不过CI也有其代价：你必须使用相关的实践以确保主线随时处于健康状态，必须学会将大功能拆分成小块，还必须使用特性开关（feature toggle，也叫特性旗标，feature flag）将尚未完成又无法拆小的功能隐藏掉。

CI的粉丝之所以喜欢这种工作方式，部分原因

是它降低了分支合并的难度，不过最重要的原因还是CI与重构能良好配合。重构经常需要对代码库中的很多地方做很小的修改（例如给一个广泛使用的函数改名），这样的修改尤其容易造成合并时的语义冲突。采用特性分支的团队常会发现重构加剧了分支合并的困难，并因此放弃了重构，这种情况我们曾经见过多次。CI和重构能够良好配合，所以Kent Beck在极限编程中同时包含了这两个实践。

我并不是在说绝不应该使用特性分支。如果特性分支存在的时间足够短，它们就不会造成大问题。（实际上，使用CI的团队往往同时也使用分支，但他们会每天将分支与主线合并。）对于开源项目，特性分支可能是合适的做法，因为不时会有你不熟悉（因此也不信任）的程序员偶尔提交修改。但对全职的开发团队而言，特性分支对重构的阻碍太严重了。即便你没有完全采用CI，我也一定会催促你尽可能频繁地集成。而且，用上CI的团队在软件交付上更加高效，我真心希望你认真考虑这个客观事实[Forsgren et al]。

测试

不会改变程序可观察的行为，这是重构的一个

重要特征。如果仔细遵循重构手法的每个步骤，我应该不会破坏任何东西，但万一我犯了个错误怎么办？（呃，就我这个粗心大意的性格来说，请去掉“万一”两字。）人总会有出错的时候，不过只要及时发现，就不会造成大问题。既然每个重构都是很小的修改，即便真的造成了破坏，我也只需要检查最后一步的小修改——就算找不到出错的原因，只要回滚到版本控制中最后一个可用的版本就行了。

这里的关键就在于“快速发现错误”。要做到这一点，我的代码应该有一套完备的测试套件，并且运行速度要快，否则我会不愿意频繁运行它。也就是说，绝大多数情况下，如果想要重构，我得先有可以自测试的代码[mf-stc]。

有些读者可能会觉得，“自测试的代码”这个要求太高，根本无法实现。但在过去20年中，我看到很多团队以这种方式构造软件。的确，团队必须投入时间与精力在测试上，但收益是绝对划算的。自测试的代码不仅使重构成为可能，而且使添加新功能更加安全，因为我可以很快发现并干掉新近引入的bug。这里的关键在于，一旦测试失败，我只需要查看上次测试成功运行之后修改的这部分代码；如果测试运行得很频繁，这个查看的范围就只有几行代码。知道必定是这几行代码造成bug的话，排

查起来会容易得多。

这也回答了“重构风险太大，可能引入bug”的担忧。如果没有自测试的代码，这种担忧就是完全合理的，这也是为什么我如此重视可靠的测试。

缺乏测试的问题可以用另一种方式来解决。如果我的开发环境很好地支持自动化重构，我就可以信任这些重构，不必运行测试。这时即便没有完备的测试套件，我仍然可以重构，前提是仅仅使用那些自动化的、一定安全的重构手法。这会让我损失很多好用的重构手法，不过剩下可用的也不少，我还是能从中获益。当然，我还是更愿意有自测试的代码，但如果没有，自动化重构的工具包也很好。

缺乏测试的现状还催生了另一种重构的流派：只使用一组经过验证是安全的重构手法。这个流派要求严格遵循重构的每个步骤，并且可用的重构手法是特定于语言的。使用这种方法，团队得以在测试覆盖率很低的大型代码库上开展一些有用的重构。这个重构流派比较新，涉及一些很具体、特定于编程语言的技巧与做法，行业里对这种方法的介绍和了解都还不足，因此本书不对其多做介绍。（不过我希望未来在我自己的网站上多讨论这个主题。感兴趣的读者可以查看Jay Bazuzi关于如何在C++中安全地运用提炼函数（106）的描述[Bazuzi]，借此获得一点儿对这个重构流派的了

解。）

毫不意外，自测试代码与持续集成紧密相关——我们仰赖持续集成来及时捕获分支集成时的语义冲突。自测试代码是极限编程的另一个重要组成部分，也是持续交付的关键环节。

遗留代码

大多数人会觉得，有一大笔遗产是件好事，但从程序员的角度来看就不同了。遗留代码往往很复杂，测试又不足，而且最关键的是，是别人写的（瑟瑟发抖）。

重构可以很好地帮助我们理解遗留系统。引人误解的函数名可以改名，使其更好地反映代码用途；糟糕的程序结构可以慢慢理顺，把程序从一块顽石打磨成美玉。整个故事都很棒，但我们绕不开关底的恶龙：遗留系统多半没测试。如果你面对一个庞大而又缺乏测试的遗留系统，很难安全地重构清理它。

对于这个问题，显而易见的答案是“没测试就加测试”。这事听起来简单（当然工作量必定很大），操作起来可没那么容易。一般来说，只有在

设计系统时就考虑到了测试，这样的系统才容易添加测试——可要是如此，系统早该有测试了，我也不用操这份心了。

这个问题没有简单的解决办法，我能给出的最好建议就是买一本《修改代码的艺术》[Feathers]，照书里的指导来做。别担心那本书太老，尽管已经出版十多年，其中的建议仍然管用。一言以蔽之，它建议你先找到程序的接缝，在接缝处插入测试，如此将系统置于测试覆盖之下。你需要运用重构手法创造出接缝——这样的重构很危险，因为没有测试覆盖，但这是为了取得进展必要的风险。在这种情况下，安全的自动化重构简直就是天赐福音。如果这一切听起来很困难，因为它确实很困难。很遗憾，一旦跌进这个深坑，没有爬出来的捷径，这也是我强烈倡导从一开始就写能自测试的代码的原因。

就算有了测试，我也不建议你尝试一鼓作气把复杂而混乱的遗留代码重构成漂亮的代码。我更愿意随时重构相关的代码：每次触碰一块代码时，我会尝试把它变好一点点——至少要让营地比我到达时更干净。如果是一个大系统，越是频繁使用的代码，改善其可理解性的努力就能得到越丰厚的回报。

数据库

在本书的第1版中，我说过数据库是“重构经常出问题的一个领域”。然而在第1版问世之后仅仅一年，情况就发生了改变：我的同事Pramod Sadalage发展出一套渐进式数据库设计[mf-evodb]和数据库重构[Ambler & Sadalage]的办法，如今已经被广泛使用。这项技术的精要在于：借助数据迁移脚本，将数据库结构的修改与代码相结合，使大规模的、涉及数据库的修改可以比较容易地开展。

假设我们要对一个数据库字段（列）改名。和改变函数声明（124）一样，我要找出结构的声明处和所有调用处，然后一次完成所有修改。但这里的复杂之处在于，原来基于旧字段的数据，也要转为使用新字段。我会写一小段代码来执行数据转化的逻辑，并把这段代码放进版本控制，跟数据结构声明与使用代码的修改一并提交。此后如果我想把数据库迁移到某个版本，只要执行当前数据库版本与目标版本之间的所有迁移脚本即可。

跟通常的重构一样，数据库重构的关键也是小步修改并且每次修改都应该完整，这样每次迁移之后系统仍然能运行。由于每次迁移涉及的修改都很小，写起来应该容易；将多个迁移串联起来，就能

对数据库结构及其中存储的数据做很大的调整。

与常规的重构不同，很多时候，数据库重构最好是分散到多次生产发布来完成，这样即便某次修改在生产数据库上造成了问题，也比较容易回滚。比如，要改名一个字段，我的第一次提交会新添一个字段，但暂时不使用它。然后我会修改数据写入的逻辑，使其同时写入新旧两个字段。随后我就可以修改读取数据的地方，将它们逐个改为使用新字段。这步修改完成之后，我会暂停一小段时间，看看是否有bug冒出来。确定没有bug之后，我再删除已经没人使用的旧字段。这种修改数据库的方式是并行修改（Parallel Change，也叫扩展协议/expand-contract）[mf-pc]的一个实例。

2.6　重构、架构和YAGNI

重构极大地改变了人们考虑软件架构的方式。在我的职业生涯早期，我被告知：在任何人开始写代码之前，必须先完成软件的设计和架构。一旦代码写出来，架构就固定了，只会因为程序员的草率对待而逐渐腐败。

重构改变了这种观点。有了重构技术，即便是已经在生产环境中运行了多年的软件，我们也有能力大幅度修改其架构。正如本书的副标题所指出的，重构可以改善既有代码的设计。但我在前面也提到了，修改遗留代码经常很有挑战，尤其当遗留代码缺乏恰当的测试时。

重构对架构最大的影响在于，通过重构，我们能得到一个设计良好的代码库，使其能够优雅地应对不断变化的需求。“在编码之前先完成架构”这种做法最大的问题在于，它假设了软件的需求可以预先充分理解。但经验显示，这个假设很多时候甚至可以说大多数时候是不切实际的。只有真正使用了软件、看到了软件对工作的影响，人们才会想明白自己到底需要什么，这样的例子不胜枚举。

应对未来变化的办法之一，就是在软件里植入

灵活性机制。在编写一个函数时，我会考虑它是否有更通用的用途。为了应对我预期的应用场景，我预测可以给这个函数加上十多个参数。这些参数就是灵活性机制——跟大多数“机制”一样，它不是免费午餐。把所有这些参数都加上的话，函数在当前的使用场景下就会非常复杂。另外，如果我少考虑了一个参数，已经加上的这一堆参数会使新添参数更麻烦。而且我经常会把灵活性机制弄错——可能是未来的需求变更并非以我期望的方式发生，也可能我对机制的设计不好。考虑到所有这些因素，很多时候这些灵活性机制反而拖慢了我响应变化的速度。

有了重构技术，我就可以采取不同的策略。与其猜测未来需要哪些灵活性、需要什么机制来提供灵活性，我更愿意只根据当前的需求来构造软件，同时把软件的设计质量做得很高。随着对用户需求的理解加深，我会对架构进行重构，使其能够应对新的需要。如果一种灵活性机制不会增加复杂度（比如添加几个命名良好的小函数），我可以很开心地引入它；但如果一种灵活性会增加软件复杂度，就必须先证明自己值得被引入。如果不同的调用者不会传入不同的参数值，那么就不要添加这个参数。当真的需要添加这个参数时，运用函数参数化（310）也很容易。要判断是否应该为未来的变化添加灵活性，我会评估“如果以后再重构有多困

难”，只有当未来重构会很困难时，我才考虑现在就添加灵活性机制。我发现这是一个很有用的决策方法。

这种设计方法有很多名字：简单设计、增量式设计或者YAGNI[mf-yagni]——“你不会需要它”（you aren’t going to need it）的缩写。YAGNI并不是“不做架构性思考”的意思，不过确实有人以这种欠考虑的方式做事。我把YAGNI视为将架构、设计与开发过程融合的一种工作方式，这种工作方式必须有重构作为基础才可靠。

采用YAGNI并不表示完全不用预先考虑架构。总有一些时候，如果缺少预先的思考，重构会难以开展。但两者之间的平衡点已经发生了很大的改变：如今我更倾向于等一等，待到对问题理解更充分，再来着手解决。演进式架构[Ford et al.]是一门仍在不断发展的学科，架构师们在不断探索有用的模式和实践，充分发挥迭代式架构决策的能力。

2.7 重构与软件开发过程

读完前面“重构的挑战”一节，你大概已经有这个印象：重构是否有效，与团队采用的其他软件开发实践紧密相关。重构起初是作为极限编程（XP）[mf-xp]的一部分被人们采用的，XP本身就融合了一组不太常见而又彼此关联的实践，例如持续集成、自测试代码以及重构（后两者融汇成了测试驱动开发）。

极限编程是最早的敏捷软件开发方法[mf-nm]之一。在一段历史时期，极限编程引领了敏捷的崛起。如今已经有很多项目使用敏捷方法，甚至敏捷的思维已经被视为主流，但实际上大部分“敏捷”项目只是徒有其名。要真正以敏捷的方式运作项目，团队成员必须在重构上有能力、有热情，他们采用的开发过程必须与常规的、持续的重构相匹配。

重构的第一块基石是自测试代码。我应该有一套自动化的测试，我可以频繁地运行它们，并且我有信心：如果我在编程过程中犯了任何错误，会有测试失败。这块基石如此重要，我会专门用一章篇幅来讨论它。

如果一支团队想要重构，那么每个团队成员都

需要掌握重构技能，能在需要时开展重构，而不会干扰其他人的工作。这也是我鼓励持续集成的原因：有了CI，每个成员的重构都能快速分享给其他同事，不会发生这边在调用一个接口那边却已把这个接口删掉的情况；如果一次重构会影响别人的工作，我们很快就会知道。自测试的代码也是持续集成的关键环节，所以这三大实践——自测试代码、持续集成、重构——彼此之间有着很强的协同效应。

有这三大实践在手，我们就能运用前一节介绍的YAGNI设计方法。重构和YAGNI交相呼应、彼此增效，重构（及其前置实践）是YAGNI的基础，YAGNI又让重构更易于开展：比起一个塞满了想当然的灵活性的系统，当然是修改一个简单的系统要容易得多。在这些实践之间找到合适的平衡点，你就能进入良性循环，你的代码既牢固可靠又能快速响应变化的需求。

有这三大核心实践打下的基础，才谈得上运用敏捷思想的其他部分。持续交付确保软件始终处于可发布的状态，很多互联网团队能做到一天多次发布，靠的正是持续交付的威力。即便我们不需要如此频繁的发布，持续集成也能帮我们降低风险，并使我们做到根据业务需要随时安排发布，而不受技术的局限。有了可靠的技术根基，我们能够极大地

压缩“从好点子到生产代码”的周期时间，从而更好地服务客户。这些技术实践也会增加软件的可靠性，减少耗费在bug上的时间。

这一切说起来似乎很简单，但实际做起来毫不容易。不管采用什么方法，软件开发都是一件复杂而微妙的事，涉及人与人之间、人与机器之间的复杂交互。我在这里描述的方法已经被证明可以应对这些复杂性，但——就跟其他所有方法一样——对使用者的实践和技能有要求。

2.8　重构与性能

关于重构，有一个常被提出的问题：它对程序的性能将造成怎样的影响？为了让软件易于理解，我常会做出一些使程序运行变慢的修改。这是一个重要的问题。我并不赞成为了提高设计的纯洁性而忽视性能，把希望寄托于更快的硬件身上也绝非正道。已经有很多软件因为速度太慢而被用户拒绝，日益提高的机器速度也只不过略微放宽了速度方面的限制而已。但是，换个角度说，虽然重构可能使软件运行更慢，但它也使软件的性能优化更容易。除了对性能有严格要求的实时系统，其他任何情况下“编写快速软件”的秘密就是：先写出可调优的软件，然后调优它以求获得足够的速度。

我看过3种编写快速软件的方法。其中最严格的是时间预算法，这通常只用于性能要求极高的实时系统。如果使用这种方法，分解你的设计时就要做好预算，给每个组件预先分配一定资源，包括时间和空间占用。每个组件绝对不能超出自己的预算，就算拥有组件之间调度预配时间的机制也不行。这种方法高度重视性能，对于心律调节器一类的系统是必需的，因为在这样的系统中迟来的数据就是错误的数据。但对其他系统（例如我经常开发

的企业信息系统）而言，如此追求高性能就有点儿过分了。

第二种方法是持续关注法。这种方法要求任何程序员在任何时间做任何事时，都要设法保持系统的高性能。这种方式很常见，感觉上很有吸引力，但通常不会起太大作用。任何修改如果是为了提高性能，通常会使程序难以维护，继而减缓开发速度。如果最终得到的软件的确更快了，那么这点损失尚有所值，可惜通常事与愿违，因为性能改善一旦被分散到程序各个角落，每次改善都只不过是从对程序行为的一个狭隘视角出发而已，而且常常伴随着对编译器、运行时环境和硬件行为的误解。

劳而无获

> 克莱斯勒综合薪资系统的支付过程太慢了。虽然我们的开发还没结束，这个问题却已经开始困扰我们，因为它已经拖累了测试速度。
>
> Kent Beck、Martin Fowler和我决定解决这个问题。等待大伙儿会合的时间里，凭着对这个系统的全盘了解，我开始推测：到底是什么让系统变慢了？我想到数种可能，然后和伙伴们谈了几种可能的修改方案。最后，我们就“如何让这个系统运行更快”，提出了一些真正的好点子。
>
> 然后，我们拿Kent的工具度量了系统性能。我一开始所想的可能性竟然全都不是问题肇因。我们发现：系统把一半时间用来创建“日期”实例（instance）。更有趣的是，所有这些实例都有相同的几个值。

于是我们观察日期对象的创建逻辑，发现有机会将它优化。这些日期对象在创建时都经过了一个字符串转换过程，然而这里并没有任何外部数据输入。之所以使用字符串转换方式，完全只是因为代码写起来简单。好，也许我们可以优化它。

然后，我们观察这些日期对象是如何被使用的。我们发现，很多日期对象都被用来产生“日期区间”实例——由一个起始日期和一个结束日期组成的对象。仔细追踪下去，我们发现绝大多数日期区间是空的！

处理日期区间时我们遵循这样一个规则：如果结束日期在起始日期之前，这个日期区间就该是空的。这是一条很好的规则，完全符合这个类的需要。采用此规则后不久，我们意识到，创建一个“起始日期在结束日期之后”的日期区间，仍然不算是清晰的代码，于是我们把这个行为提炼成一个工厂函数，由它专门创建“空的日期区间”。

我们做了上述修改，使代码更加清晰，也意外得到了一个惊喜：可以创建一个固定不变的“空日期区间”对象，并让上述调整后的工厂函数始终返回该对象，而不再每次都创建新对象。这一修改把系统速度提升了几乎一倍，足以让测试速度达到可接受的程度。这只花了我们大约五分钟。

我和团队成员（Kent和Martin谢绝参加）认真推测过：我们了若指掌的这个程序中可能有什么错误？我们甚至凭空做了些改进设计，却没有先对系统的真实情况进行度量。

我们完全错了。除了一场很有趣的交谈，我们什么好事都没做。

教训是：哪怕你完全了解系统，也请实际度量它的性能，不要臆测。臆测会让你学到一些东西，但十有八九你是错的。

——Ron Jeffries

关于性能，一件很有趣的事情是：如果你对大多数程序进行分析，就会发现它把大半时间都耗费

在一小半代码身上。如果你一视同仁地优化所有代码，90％的优化工作都是白费劲的，因为被你优化的代码大多很少被执行。你花时间做优化是为了让程序运行更快，但如果因为缺乏对程序的清楚认识而花费时间，那些时间就都被浪费掉了。

第三种性能提升法就是利用上述的90%统计数据。采用这种方法时，我编写构造良好的程序，不对性能投以特别的关注，直至进入性能优化阶段——那通常是在开发后期。一旦进入该阶段，我再遵循特定的流程来调优程序性能。

在性能优化阶段，我首先应该用一个度量工具来监控程序的运行，让它告诉我程序中哪些地方大量消耗时间和空间。这样我就可以找出性能热点所在的一小段代码。然后我应该集中关注这些性能热点，并使用持续关注法中的优化手段来优化它们。由于把注意力都集中在热点上，较少的工作量便可显现较好的成果。即便如此，我还是必须保持谨慎。和重构一样，我会小幅度进行修改。每走一步都需要编译、测试，再次度量。如果没能提高性能，就应该撤销此次修改。我会继续这个“发现热点，去除热点”的过程，直到获得客户满意的性能为止。

一个构造良好的程序可从两方面帮助这一优化方式。首先，它让我有比较充裕的时间进行性能调

整，因为有构造良好的代码在手，我能够更快速地添加功能，也就有更多时间用在性能问题上（准确的度量则保证我把这些时间投在恰当地点）。其次，面对构造良好的程序，我在进行性能分析时便有较细的粒度。度量工具会把我带入范围较小的代码段中，而性能的调整也比较容易些。由于代码更加清晰，因此我能够更好地理解自己的选择，更清楚哪种调整起关键作用。

我发现重构可以帮助我写出更快的软件。短期看来，重构的确可能使软件变慢，但它使优化阶段的软件性能调优更容易，最终还是会得到好的效果。

2.9 重构起源何处

我曾经努力想找出“重构”（refactoring）一词的真正起源，但最终失败了。优秀程序员肯定至少会花一些时间来清理自己的代码。这么做是因为，他们知道整洁的代码比杂乱无章的代码更容易修改，而且他们知道自己几乎无法一开始就写出整洁的代码。

重构不止如此。本书中我把重构看作整个软件开发过程的一个关键环节。最早认识重构重要性的两个人是Ward Cunningham和Kent Beck，他们早在20世纪80年代就开始使用Smalltalk，那是一个特别适合重构的环境。Smalltalk是一个十分动态的环境，用它可以很快写出功能丰富的软件。Smalltalk的“编译-链接-执行”周期非常短，因此很容易快速修改代码——要知道，当时很多编程环境做一次编译就需要整晚时间。它支持面向对象，也有强大的工具，最大限度地将修改的影响隐藏于定义良好的接口背后。Ward和Kent努力探索出一套适合这类环境的软件开发过程（如今，Kent把这种风格叫作极限编程）。他们意识到：重构对于提高生产力非常重要。从那时起他们就一直在工作中运用重构技术，在正式的软件项目中使用它，并不断精炼重构

的过程。

Ward和Kent的思想对Smalltalk社区产生了极大影响，重构概念也成为Smalltalk文化中的一个重要元素。Smalltalk社区的另一位领袖是Ralph Johnson，伊利诺伊大学厄巴纳-香槟分校教授，著名的GoF[gof]之一。Ralph最大的兴趣之一就是开发软件框架。他揭示了重构有助于灵活高效框架的开发。

Bill Opdyke是Ralph的博士研究生，对框架也很感兴趣。他看到了重构的潜在价值，并看到重构应用于Smalltalk之外的其他语言的可能性。他的技术背景是电话交换系统的开发。在这种系统中，大量的复杂情况与日俱增，而且非常难以修改。Bill的博士研究就是从工具构筑者的角度来看待重构。Bill对C++的框架开发中用得上的重构手法特别感兴趣。他也研究了极有必要的“语义保持的重构”（semantics-preserving refactoring），并阐明了如何证明这些重构是语义保持的，以及如何用工具实现重构。Bill的博士论文[Opdyke]是重构领域中第一部丰硕的研究成果。

我还记得1992年OOPSLA大会上见到Bill的情景。我们坐在一间咖啡厅里，Bill跟我谈起他的研究成果，我还记得自己当时的想法：“有趣，但并非真的那么重要。”唉，我完全错了。

John Brant和Don Roberts将“重构工具”的构想发扬光大，开发了一个名为Refactoring Browser（重构浏览器）的重构工具。这是第一个自动化的重构工具，多亏Smalltalk提供了适合重构的编程环境。

那么，我呢？我一直有清理代码的倾向，但从来没有想到这会如此重要。后来我和Kent一起做一个项目，看到他使用重构手法，也看到重构对开发效能和质量带来的影响。这份体验让我相信：重构是一门非常重要的技术。但是，在重构的学习和推广过程中我遇到了挫折，因为我拿不出任何一本书给程序员看，也没有任何一位专家打算写这样一本书。所以，在这些专家的帮助下，我写下了这本书的第1版。

幸运的是，重构的概念被行业广泛接受了。本书第1版销量不错，“重构”一词也走进了大多数程序员的词汇库。更多的重构工具涌现出来，尤其是在Java世界里。重构的流行也带来了负面效应：很多人随意地使用“重构”这个词，而他们真正做的却是不严谨的结构调整。尽管如此，重构终归成了一项主流的软件开发实践。

2.10　自动化重构

过去10年中，重构领域最大的变化可能就是出现了一批支持自动化重构的工具。如果我想给一个Java的方法改名，在IntelliJ IDEA或者Eclipse这样的开发环境中，我只需要从菜单里点选对应的选项，工具会帮我完成整个重构过程，而且我通常都可以相信，工具完成的重构是可靠的，所以用不着运行测试套件。

第一个自动化重构工具是Smalltalk的Refactoring Browser，由John Brandt和Don Roberts开发。在21世纪初，Java世界的自动化重构工具如雨后春笋般涌现。在JetBrains的IntelliJ IDEA集成开发环境（IDE）中，自动化重构是最亮眼的特性之一。IBM也紧随其后，在VisualAge的Java版中也提供了重构工具。VisualAge的影响力有限，不过其中很多能力后来被Eclipse继承，包括对重构的支持。

重构也进入了C#世界，起初是通过JetBrains的Resharper，这是一个Visual Studio插件。后来Visual Studio团队直接在IDE里提供了一些重构能力。

如今的编辑器和开发工具中常能找到一些对重构的支持，不过真实的重构能力各有高低。重构能

力的差异既有工具的原因，也受限于不同语言对自动化重构的支持程度。在这里，我不打算分析各种工具的能力，不过谈谈重构工具背后的原则还是有点儿意思的。

一种粗糙的自动化重构方式是文本操作，比如用查找/替换的方式给函数改名，或者完成提炼变量（119）所需的简单结构调整。这种方法太粗糙了，做完之后必须重新运行测试，否则不能信任。但这可以是一个便捷的起步。在用Emacs编程时，没有那些更完善的重构支持，我也会用类似的文本操作宏来加速重构。

要支持体面的重构，工具只操作代码文本是不行的，必须操作代码的语法树，这样才能更可靠地保持代码行为。所以，今天的大多数重构功能都依附于强大的IDE，因为这些IDE原本就在语法树上实现了代码导航、静态检查等功能，自然也可以用于重构。不仅能处理文本，还能处理语法树，这是IDE相比于文本编辑器更先进的地方。

重构工具不仅需要理解和修改语法树，还要知道如何把修改后的代码写回编辑器视图。总而言之，实现一个体面的自动化重构手法，是一个很有挑战的编程任务。尽管我一直开心地使用重构工具，对它们背后的实现却知之甚少。

在静态类型语言中，很多重构手法会更加安全。假设我想做一次简单的函数改名（124）：在Salesman类和Server类中都有一个叫作addClient的函数，当然两者各有其用途。我想对Salesman中的addClient函数改名，Server类中的函数则保持不变。如果不是静态类型，工具很难识别调用addClient的地方到底是在使用哪个类的函数。Smalltalk的Refactoring Browser会列出所有调用点，我需要手工决定修改哪些调用点。这个重构是不安全的，我必须重新运行所有测试。这样的工具仍然有用，但在Java中的函数改名（124）重构则可以是完全安全、完全自动的，因为在静态类型的帮助下，工具可以识别函数所属的类，所以它只会修改应该修改的那些函数调用点，对此我可以完全放心。

一些重构工具走得更远。如果我给一个变量改名，工具会提醒我修改使用了旧名字的注释。如果我使用提炼函数（106），工具会找出与新函数体重复的代码片段，建议代之以对新函数的调用。在编程时可以使用如此强大的重构功能，这就是为什么我们要使用一个体面的IDE，而不是固执于熟悉的文本编辑器。我个人很喜欢用Emacs，但在使用Java时，我更愿意用IntelliJ IDEA或者Eclipse，很大程度上就是为了获得重构支持。

尽管这些强大的重构工具有着魔法般的能力，可以安全地重构代码，但还是会有闪失出现。通过反射进行的调用（例如Java中的`Method.invoke`）会迷惑不够成熟的重构工具，但比较成熟的工具则可以很好地应对。所以，即便是最安全的重构，也应该经常运行测试套件，以确保没有什么东西在不经意间被破坏。我经常会间杂进行自动重构和手动重构，所以运行测试的频度是足够的。

能借助语法树来分析和重构程序代码，这是IDE与普通文本编辑器相比具有的一大优势。但很多程序员又喜欢用得顺手的文本编辑器的灵活性，希望鱼与熊掌兼得。语言服务器（Language Server）是一种正在引起关注的新技术：用软件生成语法树，给文本编辑器提供API。语言服务器可以支持多种文本编辑器，并且为强大的代码分析和重构操作提供了命令。

2.11　延展阅读

在第2章就开始谈延展阅读，这似乎有点儿奇怪。不过，有大量关于重构的材料已经超出了本书的范围，早些让读者知道这些材料的存在也是件好事。

本书的第1版教很多人学会了重构，不过我的关注点是组织一本重构的参考书，而不是带领读者走过学习过程。如果你需要一本面向入门者的教材，我推荐Bill Wake的《重构手册》[Wake]，其中包含了很多有用的重构练习。

很多重构的先行者同时也活跃于软件模式社区。Josh Kerievsky在《重构与模式》[Kerievsky]一书中紧密连接了这两个世界。他审视了影响巨大的GoF[gof]书中一些最有价值的模式，并展示了如何通过重构使代码向这些模式的方向演化。

本书聚焦讨论通用编程语言中的重构技巧。还有一些专门领域的重构，例如已经引起关注的《数据库重构》[Ambler & Sadalage]（由Scott Ambler和Pramod Sadalage所著）和《重构HTML》[Harold]（由Elliotte Rusty Harold所著）。

尽管标题中没有“重构”二字，Michael Feathers的《修改代码的艺术》[Feathers]也不得不提。这本书主要讨论如何在缺乏测试覆盖的老旧代码库上开展重构。

本书（及其前一版）对读者的编程语言背景没有要求。也有人写专门针对特定语言的重构书籍。我的两位前同事Jay Fields和Shane Harvey就撰写了Ruby版的《重构》[Fields et al.]。

在本书的Web版和重构网站（refactoring.com）[ref.com]上都可以找到更多相关材料的更新。

第3章　代码的坏味道

——Kent Beck和Martin Fowler

“如果尿布臭了，就换掉它。”

——语出Beck奶奶，论保持小孩清洁的哲学

现在，对于重构如何运作，你已经有了相当好的理解。但是知道“如何”不代表知道“何时”。决定何时重构及何时停止和知道重构机制如何运转一样重要。

难题来了！解释“如何删除一个实例变量”或“如何产生一个继承体系”很容易，因为这些都是很简单的事情，但要解释“该在什么时候做这些动作”就没那么顺理成章了。除了露几手含混的编程美学（说实话，这就是咱们这些顾问常做的事），我还希望让某些东西更具说服力一些。

撰写本书的第1版时，我正在为这个微妙的问题大伤脑筋。去苏黎世拜访Kent Beck的时候，也许是因为受到刚出生的女儿的气味影响吧，他提出用味道来形容重构的时机。

“味道，”你可能会说，“真的比含混的美学理论要好吗？”好吧，是的。我们看过很多很多代码，它们所属的项目从大获成功到奄奄一息都有。观察这些代码时，我们学会了从中找寻某些特定结构，这些结构指出（有时甚至就像尖叫呼喊）重构的可能性。（本章主语换成“我们”，是为了反映一个事实：Kent和我共同撰写本章。你应该可以看出我俩的文笔差异——插科打诨的部分是我写的，其余都是他写的。）

我们并不试图给你一个何时必须重构的精确衡量标准。从我们的经验看来，没有任何量度规矩比得上见识广博者的直觉。我们只会告诉你一些迹象，它会指出“这里有一个可以用重构解决的问题”。你必须培养自己的判断力，学会判断一个类内有多少实例变量算是太大、一个函数内有多少行代码才算太长。

如果你无法确定该采用哪一种重构手法，请阅读本章内容和书后附的“重构列表”来寻找灵感。你可以阅读本章或快速浏览书后附的“坏味道与重构手法速查表”来判断自己闻到的是什么味道，然后再看看我们所建议的重构手法能否帮到你。也许这里所列的“坏味道条款”和你所检测的不尽相符，但愿它们能够为你指引正确方向。

3.1 神秘命名（Mysterious Name）

读侦探小说时，透过一些神秘的文字猜测故事情节是一种很棒的体验；但如果是在阅读代码，这样的体验就不怎么好了。我们也许会幻想自己是《王牌大贱谍》中的国际特工**1**，但我们写下的代码应该直观明了。整洁代码最重要的一环就是好的名字，所以我们会深思熟虑如何给函数、模块、变量和类命名，使它们能清晰地表明自己的功能和用法。

然而，很遗憾，命名是编程中最难的两件事之一[mf-2h]。正因为如此，改名可能是最常用的重构手法，包括改变函数声明（124）（用于给函数改名）、变量改名（137）、字段改名（244）等。很多人经常不愿意给程序元素改名，觉得不值得费这个劲，但好的名字能节省未来用在猜谜上的大把时间。

改名不仅仅是修改名字而已。如果你想不出一个好名字，说明背后很可能潜藏着更深的设计问题。为一个恼人的名字所付出的纠结，常常能推动我们对代码进行精简。

1《王牌大贱谍》（International Man of Mystery）是1997年杰伊·罗奇执导的一部喜剧谍战片。——译者注

3.2　重复代码（Duplicated Code）

如果你在一个以上的地点看到相同的代码结构，那么可以肯定：设法将它们合而为一，程序会变得更好。一旦有重复代码存在，阅读这些重复的代码时你就必须加倍仔细，留意其间细微的差异。如果要修改重复代码，你必须找出所有的副本来修改。

最单纯的重复代码就是“同一个类的两个函数含有相同的表达式”。这时候你需要做的就是采用提炼函数（106）提炼出重复的代码，然后让这两个地点都调用被提炼出来的那一段代码。如果重复代码只是相似而不是完全相同，请首先尝试用移动语句（223）重组代码顺序，把相似的部分放在一起以便提炼。如果重复的代码段位于同一个超类的不同子类中，可以使用函数上移（350）来避免在两个子类之间互相调用。

3.3　过长函数（Long Function）

据我们的经验，活得最长、最好的程序，其中的函数都比较短。初次接触到这种代码库的程序员常常会觉得“计算都没有发生”——程序里满是无穷无尽的委托调用。但和这样的程序共处几年之后，你就会明白这些小函数的价值所在。间接性带来的好处——更好的阐释力、更易于分享、更多的选择——都是由小函数来支持的。

早在编程的洪荒年代，程序员们就已认识到：函数越长，就越难理解。在早期的编程语言中，子程序调用需要额外开销，这使得人们不太乐意使用小函数。现代编程语言几乎已经完全免除了进程内的函数调用开销。固然，小函数也会给代码的阅读者带来一些负担，因为你必须经常切换上下文，才能看明白函数在做什么。但现代的开发环境让你可以在函数的调用处与声明处之间快速跳转，或是同时看到这两处，让你根本不用来回跳转。不过说到底，让小函数易于理解的关键还是在于良好的命名。如果你能给函数起个好名字，阅读代码的人就可以通过名字了解函数的作用，根本不必去看其中写了些什么。

最终的效果是：你应该更积极地分解函数。我们遵循这样一条原则：每当感觉需要以注释来说明点什么的时候，我们就把需要说明的东西写进一个独立函数中，并以其用途（而非实现手法）命名。我们可以对一组甚至短短一行代码做这件事。哪怕替换后的函数调用动作比函数自身还长，只要函数名称能够解释其用途，我们也该毫不犹豫地那么做。关键不在于函数的长度，而在于函数“做什么”和“如何做”之间的语义距离。

百分之九十九的场合里，要把函数变短，只需使用提炼函数（106）。找到函数中适合集中在一起的部分，将它们提炼出来形成一个新函数。

如果函数内有大量的参数和临时变量，它们会对你的函数提炼形成阻碍。如果你尝试运用提炼函数（106），最终就会把许多参数传递给被提炼出来的新函数，导致可读性几乎没有任何提升。此时，你可以经常运用以查询取代临时变量（178）来消除这些临时元素。引入参数对象（140）和保持对象完整（319）则可以将过长的参数列表变得更简洁一些。

如果你已经这么做了，仍然有太多临时变量和参数，那就应该使出我们的杀手锏——以命令取代函数（337）。

如何确定该提炼哪一段代码呢？一个很好的技巧是：寻找注释。它们通常能指出代码用途和实现手法之间的语义距离。如果代码前方有一行注释，就是在提醒你：可以将这段代码替换成一个函数，而且可以在注释的基础上给这个函数命名。就算只有一行代码，如果它需要以注释来说明，那也值得将它提炼到独立函数中去。

条件表达式和循环常常也是提炼的信号。你可以使用分解条件表达式（260）处理条件表达式。对于庞大的switch语句，其中的每个分支都应该通过提炼函数（106）变成独立的函数调用。如果有多个switch语句基于同一个条件进行分支选择，就应该使用以多态取代条件表达式（272）。

至于循环，你应该将循环和循环内的代码提炼到一个独立的函数中。如果你发现提炼出的循环很难命名，可能是因为其中做了几件不同的事。如果是这种情况，请勇敢地使用拆分循环（227）将其拆分成各自独立的任务。

3.4　过长参数列表（Long Parameter List）

刚开始学习编程的时候，老师教我们：把函数所需的所有东西都以参数的形式传递进去。这可以理解，因为除此之外就只能选择全局数据，而全局数据很快就会变成邪恶的东西。但过长的参数列表本身也经常令人迷惑。

如果可以向某个参数发起查询而获得另一个参数的值，那么就可以使用以查询取代参数（324）去掉这第二个参数。如果你发现自己正在从现有的数据结构中抽出很多数据项，就可以考虑使用保持对象完整（319）手法，直接传入原来的数据结构。如果有几项参数总是同时出现，可以用引入参数对象（140）将其合并成一个对象。如果某个参数被用作区分函数行为的标记（flag），可以使用移除标记参数（314）。

使用类可以有效地缩短参数列表。如果多个函数有同样的几个参数，引入一个类就尤为有意义。你可以使用函数组合成类（144），将这些共同的参数变成这个类的字段。如果戴上函数式编程的帽子，我们会说，这个重构过程创造了一组部分应用

函数（partially applied function）。

3.5　全局数据（Global Data）

刚开始学软件开发时，我们就听说过关于全局数据的惊悚故事——它们是如何被来自地狱第四层的恶魔发明出来，胆敢使用它们的程序员如今在何处安息。就算这些烈焰与硫黄的故事不那么可信，全局数据仍然是最刺鼻的坏味道之一。全局数据的问题在于，从代码库的任何一个角落都可以修改它，而且没有任何机制可以探测出到底哪段代码做出了修改。一次又一次，全局数据造成了那些诡异的bug，而问题的根源却在遥远的别处，想要找到出错的代码难于登天。全局数据最显而易见的形式就是全局变量，但类变量和单例（singleton）也有这样的问题。

首要的防御手段是封装变量（132），每当我们看到可能被各处的代码污染的数据，这总是我们应对的第一招。你把全局数据用一个函数包装起来，至少你就能看见修改它的地方，并开始控制对它的访问。随后，最好将这个函数（及其封装的数据）搬移到一个类或模块中，只允许模块内的代码使用它，从而尽量控制其作用域。

可以被修改的全局数据尤其可憎。如果能保证

在程序启动之后就不再修改，这样的全局数据还算相对安全，不过得有编程语言提供这样的保证才行。

全局数据印证了帕拉塞尔斯的格言：良药与毒药的区别在于剂量。有少量的全局数据或许无妨，但数量越多，处理的难度就会指数上升。即便只是少量的数据，我们也愿意将它封装起来，这是在软件演进过程中应对变化的关键所在。

3.6　可变数据（Mutable Data）

对数据的修改经常导致出乎意料的结果和难以发现的bug。我在一处更新数据，却没有意识到软件中的另一处期望着完全不同的数据，于是一个功能失效了——如果故障只在很罕见的情况下发生，要找出故障原因就会更加困难。因此，有一整个软件开发流派——函数式编程——完全建立在“数据永不改变”的概念基础上：如果要更新一个数据结构，就返回一份新的数据副本，旧的数据仍保持不变。

不过这样的编程语言仍然相对小众，大多数程序员使用的编程语言还是允许修改变量值的。即便如此，我们也不应该忽视不可变性带来的优势——仍然有很多办法可以用于约束对数据的更新，降低其风险。

可以用封装变量（132）来确保所有数据更新操作都通过很少几个函数来进行，使其更容易监控和演进。如果一个变量在不同时候被用于存储不同的东西，可以使用拆分变量（240）将其拆分为各自不同用途的变量，从而避免危险的更新操作。使用移动语句（223）和提炼函数（106）尽量把逻辑

从处理更新操作的代码中搬移出来，将没有副作用的代码与执行数据更新操作的代码分开。设计API时，可以使用将查询函数和修改函数分离（306）确保调用者不会调到有副作用的代码，除非他们真的需要更新数据。我们还乐于尽早使用移除设值函数（331）——有时只是把设值函数的使用者找出来看看，就能帮我们发现缩小变量作用域的机会。

如果可变数据的值能在其他地方计算出来，这就是一个特别刺鼻的坏味道。它不仅会造成困扰、bug和加班，而且毫无必要。消除这种坏味道的办法很简单，使用以查询取代派生变量（248）即可。

如果变量作用域只有几行代码，即使其中的数据可变，也不是什么大问题；但随着变量作用域的扩展，风险也随之增大。可以用函数组合成类（144）或者函数组合成变换（149）来限制需要对变量进行修改的代码量。如果一个变量在其内部结构中包含了数据，通常最好不要直接修改其中的数据，而是用将引用对象改为值对象（252）令其直接替换整个数据结构。

3.7　发散式变化（Divergent Change）

我们希望软件能够更容易被修改——毕竟软件本来就该是“软”的。一旦需要修改，我们希望能够跳到系统的某一点，只在该处做修改。如果不能做到这一点，你就嗅出两种紧密相关的刺鼻味道中的一种了。

如果某个模块经常因为不同的原因在不同的方向上发生变化，发散式变化就出现了。当你看着一个类说：“呃，如果新加入一个数据库，我必须修改这3个函数；如果新出现一种金融工具，我必须修改这4个函数。”这就是发散式变化的征兆。数据库交互和金融逻辑处理是两个不同的上下文，将它们分别搬移到各自独立的模块中，能让程序变得更好：每当要对某个上下文做修改时，我们只需要理解这个上下文，而不必操心另一个。“每次只关心一个上下文”这一点一直很重要，在如今这个信息爆炸、脑容量不够用的年代就愈发紧要。当然，往往只有在加入新数据库或新金融工具后，你才能发现这个坏味道。在程序刚开发出来还在随着软件系统的能力不断演进时，上下文边界通常不是那么清

晰。

如果发生变化的两个方向自然地形成了先后次序（比如说，先从数据库取出数据，再对其进行金融逻辑处理），就可以用拆分阶段（154）将两者分开，两者之间通过一个清晰的数据结构进行沟通。如果两个方向之间有更多的来回调用，就应该先创建适当的模块，然后用搬移函数（198）把处理逻辑分开。如果函数内部混合了两类处理逻辑，应该先用提炼函数（106）将其分开，然后再做搬移。如果模块是以类的形式定义的，就可以用提炼类（182）来做拆分。

3.8 霰弹式修改（Shotgun Surgery）

霰弹式修改类似于发散式变化，但又恰恰相反。如果每遇到某种变化，你都必须在许多不同的类内做出许多小修改，你所面临的坏味道就是霰弹式修改。如果需要修改的代码散布四处，你不但很难找到它们，也很容易错过某个重要的修改。

这种情况下，你应该使用搬移函数（198）和搬移字段（207）把所有需要修改的代码放进同一个模块里。如果有很多函数都在操作相似的数据，可以使用函数组合成类（144）。如果有些函数的功能是转化或者充实数据结构，可以使用函数组合成变换（149）。如果一些函数的输出可以组合后提供给一段专门使用这些计算结果的逻辑，这种时候常常用得上拆分阶段（154）。

面对霰弹式修改，一个常用的策略就是使用与内联（inline）相关的重构——如内联函数（115）或是内联类（186）——把本不该分散的逻辑拽回一处。完成内联之后，你可能会闻到过长函数或者过大的类的味道，不过你总可以用与提炼相关的重构手法将其拆解成更合理的小块。即便如此钟爱小

型的函数和类，我们也并不担心在重构的过程中暂时创建一些较大的程序单元。

3.9 依恋情结（Feature Envy）

所谓模块化，就是力求将代码分出区域，最大化区域内部的交互、最小化跨区域的交互。但有时你会发现，一个函数跟另一个模块中的函数或者数据交流格外频繁，远胜于在自己所处模块内部的交流，这就是依恋情结的典型情况。无数次经验里，我们看到某个函数为了计算某个值，从另一个对象那儿调用几乎半打的取值函数。疗法显而易见：这个函数想跟这些数据待在一起，那就使用搬移函数（198）把它移过去。有时候，函数中只有一部分受这种依恋之苦，这时候应该使用提炼函数（106）把这一部分提炼到独立的函数中，再使用搬移函数（198）带它去它的梦想家园。

当然，并非所有情况都这么简单。一个函数往往会用到几个模块的功能，那么它究竟该被置于何处呢？我们的原则是：判断哪个模块拥有的此函数使用的数据最多，然后就把这个函数和那些数据摆在一起。如果先以提炼函数（106）将这个函数分解为数个较小的函数并分别置放于不同地点，上述步骤也就比较容易完成了。

有几个复杂精巧的模式破坏了这条规则。说起

这个话题，GoF[gof]的策略（Strategy）模式和访问者（Visitor）模式立刻跳入我的脑海，Kent Beck的Self Delegation模式[Beck SBPP]也在此列。使用这些模式是为了对抗发散式变化这一坏味道。最根本的原则是：将总是一起变化的东西放在一块儿。数据和引用这些数据的行为总是一起变化的，但也有例外。如果例外出现，我们就搬移那些行为，保持变化只在一地发生。策略模式和和访问者模式使你得以轻松修改函数的行为，因为它们将少量需被覆写的行为隔离开来——当然也付出了“多一层间接性”的代价。

3.10　数据泥团（Data Clumps）

数据项就像小孩子，喜欢成群结队地待在一块儿。你常常可以在很多地方看到相同的三四项数据：两个类中相同的字段、许多函数签名中相同的参数。这些总是绑在一起出现的数据真应该拥有属于它们自己的对象。首先请找出这些数据以字段形式出现的地方，运用提炼类（182）将它们提炼到一个独立对象中。然后将注意力转移到函数签名上，运用引入参数对象（140）或保持对象完整（319）为它瘦身。这么做的直接好处是可以将很多参数列表缩短，简化函数调用。是的，不必在意数据泥团只用上新对象的一部分字段，只要以新对象取代两个（或更多）字段，就值得这么做。

一个好的评判办法是：删掉众多数据中的一项。如果这么做，其他数据有没有因而失去意义？如果它们不再有意义，这就是一个明确信号：你应该为它们产生一个新对象。

我们在这里提倡新建一个类，而不是简单的记录结构，因为一旦拥有新的类，你就有机会让程序散发出一种芳香。得到新的类以后，你就可以着手寻找“依恋情结”，这可以帮你指出能够移至新类中

的种种行为。这是一种强大的动力：有用的类被创建出来，大量的重复被消除，后续开发得以加速，原来的数据泥团终于在它们的小社会中充分发挥价值。

3.11　基本类型偏执（Primitive Obsession）

大多数编程环境都大量使用基本类型，即整数、浮点数和字符串等。一些库会引入一些小对象，如日期。但我们发现一个很有趣的现象：很多程序员不愿意创建对自己的问题域有用的基本类型，如钱、坐标、范围等。于是，我们看到了把钱当作普通数字来计算的情况、计算物理量时无视单位（如把英寸与毫米相加）的情况以及大量类似`if (a < upper && a > lower)`这样的代码。

字符串是这种坏味道的最佳培养皿，比如，电话号码不只是一串字符。一个体面的类型，至少能包含一致的显示逻辑，在用户界面上需要显示时可以使用。“用字符串来代表类似这样的数据”是如此常见的臭味，以至于人们给这类变量专门起了一个名字，叫它们“类字符串类型”（stringly typed）变量。

你可以运用以对象取代基本类型（174）将原本单独存在的数据值替换为对象，从而走出传统的洞窟，进入炙手可热的对象世界。如果想要替换的数据值是控制条件行为的类型码，则可以运用以子

类取代类型码（362）加上以多态取代条件表达式（272）的组合将它换掉。

如果你有一组总是同时出现的基本类型数据，这就是数据泥团的征兆，应该运用提炼类（182）和引入参数对象（140）来处理。

3.12 重复的switch （Repeated Switches）

如果你跟真正的面向对象布道者交谈，他们很快就会谈到switch语句的邪恶。在他们看来，任何switch语句都应该用以多态取代条件表达式（272）消除掉。我们甚至还听过这样的观点：所有条件逻辑都应该用多态取代，绝大多数if语句都应该被扫进历史的垃圾桶。

即便在不知天高地厚的青年时代，我们也从未无条件地反对条件语句。在本书第1版中，这种坏味道被称为“switch语句”（Switch Statements），那是因为在20世纪90年代末期，程序员们太过于忽视多态的价值，我们希望矫枉过正。

如今的程序员已经更多地使用多态，switch语句也不再像15年前那样有害无益，很多语言支持更复杂的switch语句，而不只是根据基本类型值来做条件判断。因此，我们现在更关注重复的switch：在不同的地方反复使用同样的switch逻辑（可能是以switch/case语句的形式，也可能是以连续的if/else语句的形式）。重复的switch的问题在

于：每当你想增加一个选择分支时，必须找到所有的switch，并逐一更新。多态给了我们对抗这种黑暗力量的武器，使我们得到更优雅的代码库。

3.13　循环语句（Loops）

从最早的编程语言开始，循环就一直是程序设计的核心要素。但我们感觉如今循环已经有点儿过时，就像喇叭裤和植绒壁纸那样。其实在撰写本书第1版的时候，我们就已经开始鄙视循环语句，但和当时的大多数编程语言一样，当时的Java还没有提供更好的替代品。如今，函数作为一等公民已经得到了广泛的支持，因此我们可以使用以管道取代循环（231）来让这些老古董退休。我们发现，管道操作（如filter和map）可以帮助我们更快地看清被处理的元素以及处理它们的动作。

3.14 冗赘的元素（Lazy Element）

程序元素（如类和函数）能给代码增加结构，从而支持变化、促进复用或者哪怕只是提供更好的名字也好，但有时我们真的不需要这层额外的结构。可能有这样一个函数，它的名字就跟实现代码看起来一模一样；也可能有这样一个类，根本就是一个简单的函数。这可能是因为，起初在编写这个函数时，程序员也许期望它将来有一天会变大、变复杂，但那一天从未到来；也可能是因为，这个类原本是有用的，但随着重构的进行越变越小，最后只剩了一个函数。不论上述哪一种原因，请让这样的程序元素庄严赴义吧。通常你只需要使用内联函数（115）或是内联类（186）。如果这个类处于一个继承体系中，可以使用折叠继承体系（380）。

3.15 夸夸其谈通用性（Speculative Generality）

这个令我们十分敏感的坏味道，命名者是Brian Foote。当有人说“噢，我想我们总有一天需要做这事”，并因而企图以各式各样的钩子和特殊情况来处理一些非必要的事情，这种坏味道就出现了。这么做的结果往往造成系统更难理解和维护。如果所有装置都会被用到，就值得那么做；如果用不到，就不值得。用不上的装置只会挡你的路，所以，把它搬开吧。

如果你的某个抽象类其实没有太大作用，请运用折叠继承体系（380）。不必要的委托可运用内联函数（115）和内联类（186）除掉。如果函数的某些参数未被用上，可以用改变函数声明（124）去掉这些参数。如果有并非真正需要、只是为不知远在何处的将来而塞进去的参数，也应该用改变函数声明（124）去掉。

如果函数或类的唯一用户是测试用例，这就飘出了坏味道“夸夸其谈通用性”。如果你发现这样的函数或类，可以先删掉测试用例，然后使用移除死代码（237）。

3.16　临时字段（Temporary Field）

有时你会看到这样的类：其内部某个字段仅为某种特定情况而设。这样的代码让人不易理解，因为你通常认为对象在所有时候都需要它的所有字段。在字段未被使用的情况下猜测当初设置它的目的，会让你发疯。

请使用提炼类（182）给这个可怜的孤儿创造一个家，然后用搬移函数（198）把所有和这些字段相关的代码都放进这个新家。也许你还可以使用引入特例（289）在“变量不合法”的情况下创建一个替代对象，从而避免写出条件式代码。

3.17　过长的消息链（Message Chains）

如果你看到用户向一个对象请求另一个对象，然后再向后者请求另一个对象，然后再请求另一个对象……这就是消息链。在实际代码中你看到的可能是一长串取值函数或一长串临时变量。采取这种方式，意味客户端代码将与查找过程中的导航结构紧密耦合。一旦对象间的关系发生任何变化，客户端就不得不做出相应修改。

这时候应该使用隐藏委托关系（189）。你可以在消息链的不同位置采用这种重构手法。理论上，你可以重构消息链上的所有对象，但这么做就会把所有中间对象都变成“中间人”。通常更好的选择是：先观察消息链最终得到的对象是用来干什么的，看看能否以提炼函数（106）把使用该对象的代码提炼到一个独立的函数中，再运用搬移函数（198）把这个函数推入消息链。如果还有许多客户端代码需要访问链上的其他对象，同样添加一个函数来完成此事。

有些人把任何函数链都视为坏东西，我们不这样想。我们的冷静镇定是出了名的，起码在这件事

上是这样的。

3.18　中间人（Middle Man）

对象的基本特征之一就是封装——对外部世界隐藏其内部细节。封装往往伴随着委托。比如，你问主管是否有时间参加一个会议，他就把这个消息“委托”给他的记事簿，然后才能回答你。很好，你没必要知道这位主管到底使用传统记事簿还是使用电子记事簿抑或是秘书来记录自己的约会。

但是人们可能过度运用委托。你也许会看到某个类的接口有一半的函数都委托给其他类，这样就是过度运用。这时应该使用移除中间人（192），直接和真正负责的对象打交道。如果这样“不干实事”的函数只有少数几个，可以运用内联函数（115）把它们放进调用端。如果这些中间人还有其他行为，可以运用以委托取代超类（399）或者以委托取代子类（381）把它变成真正的对象，这样你既可以扩展原对象的行为，又不必负担那么多的委托动作。

3.19　内幕交易（Insider Trading）

软件开发者喜欢在模块之间建起高墙，极其反感在模块之间大量交换数据，因为这会增加模块间的耦合。在实际情况里，一定的数据交换不可避免，但我们必须尽量减少这种情况，并把这种交换都放到明面上来。

如果两个模块总是在咖啡机旁边窃窃私语，就应该用搬移函数（198）和搬移字段（207）减少它们的私下交流。如果两个模块有共同的兴趣，可以尝试再新建一个模块，把这些共用的数据放在一个管理良好的地方；或者用隐藏委托关系（189），把另一个模块变成两者的中介。

继承常会造成密谋，因为子类对超类的了解总是超过后者的主观愿望。如果你觉得该让这个孩子独立生活了，请运用以委托取代子类（381）或以委托取代超类（399）让它离开继承体系。

3.20 过大的类（Large Class）

如果想利用单个类做太多事情，其内往往就会出现太多字段。一旦如此，重复代码也就接踵而至了。

你可以运用提炼类（182）将几个变量一起提炼至新类内。提炼时应该选择类内彼此相关的变量，将它们放在一起。例如，`depositAmount`和`depositCurrency`可能应该隶属同一个类。通常，如果类内的数个变量有着相同的前缀或后缀，这就意味着有机会把它们提炼到某个组件内。如果这个组件适合作为一个子类，你会发现提炼超类（375）或者以子类取代类型码（362）（其实就是提炼子类）往往比较简单。

有时候类并非在所有时刻都使用所有字段。若果真如此，你或许可以进行多次提炼。

和“太多实例变量”一样，类内如果有太多代码，也是代码重复、混乱并最终走向死亡的源头。最简单的解决方案（还记得吗，我们喜欢简单的解决方案）是把多余的东西消弭于类内部。如果有5个“百行函数”，它们之中很多代码都相同，那么或

许你可以把它们变成5个“十行函数”和10个提炼出来的“双行函数”。

观察一个大类的使用者，经常能找到如何拆分类的线索。看看使用者是否只用到了这个类所有功能的一个子集，每个这样的子集都可能拆分成一个独立的类。一旦识别出一个合适的功能子集，就试用提炼类（182）、提炼超类（375）或是以子类取代类型码（362）将其拆分出来。

3.21　异曲同工的类（Alternative Classes with Different Interfaces）

使用类的好处之一就在于可以替换：今天用这个类，未来可以换成用另一个类。但只有当两个类的接口一致时，才能做这种替换。可以用改变函数声明（124）将函数签名变得一致。但这往往还不够，请反复运用搬移函数（198）将某些行为移入类中，直到两者的协议一致为止。如果搬移过程造成了重复代码，或许可运用提炼超类（375）补偿一下。

3.22　纯数据类（Data Class）

所谓纯数据类是指：它们拥有一些字段，以及用于访问（读写）这些字段的函数，除此之外一无长物。这样的类只是一种不会说话的数据容器，它们几乎一定被其他类过分细琐地操控着。这些类早期可能拥有`public`字段，若果真如此，你应该在别人注意到它们之前，立刻运用封装记录（162）将它们封装起来。对于那些不该被其他类修改的字段，请运用移除设值函数（331）。

然后，找出这些取值/设值函数被其他类调用的地点。尝试以搬移函数（198）把那些调用行为搬移到纯数据类里来。如果无法搬移整个函数，就运用提炼函数（106）产生一个可被搬移的函数。

纯数据类常常意味着行为被放在了错误的地方。也就是说，只要把处理数据的行为从客户端搬移到纯数据类里来，就能使情况大为改观。但也有例外情况，一个最好的例外情况就是，纯数据记录对象被用作函数调用的返回结果，比如使用拆分阶段（154）之后得到的中转数据结构就是这种情况。这种结果数据对象有一个关键的特征：它是不可修改的（至少在拆分阶段（154）的实际操作中

是这样）。不可修改的字段无须封装，使用者可以直接通过字段取得数据，无须通过取值函数。

3.23　被拒绝的遗赠（Refused Bequest）

子类应该继承超类的函数和数据。但如果它们不想或不需要继承，又该怎么办呢？它们得到所有礼物，却只从中挑选几样来玩！

按传统说法，这就意味着继承体系设计错误。你需要为这个子类新建一个兄弟类，再运用函数下移（359）和字段下移（361）把所有用不到的函数下推给那个兄弟。这样一来，超类就只持有所有子类共享的东西。你常常会听到这样的建议：所有超类都应该是抽象（abstract）的。

既然使用“传统说法”这个略带贬义的词，你就可以猜到，我们不建议你这么做，起码不建议你每次都这么做。我们经常利用继承来复用一些行为，并发现这可以很好地应用于日常工作。这也是一种坏味道，我们不否认，但气味通常并不强烈，所以我们说，如果“被拒绝的遗赠”正在引起困惑和问题，请遵循传统忠告。但不必认为你每次都得那么做。十有八九这种坏味道很淡，不值得理睬。

如果子类复用了超类的行为（实现），却又不

愿意支持超类的接口，“被拒绝的遗赠”的坏味道就会变得很浓烈。拒绝继承超类的实现，这一点我们不介意；但如果拒绝支持超类的接口，这就难以接受了。既然不愿意支持超类的接口，就不要虚情假意地糊弄继承体系，应该运用以委托取代子类（381）或者以委托取代超类（399）彻底划清界限。

3.24　注释（Comments）

别担心，我们并不是说你不该写注释。从嗅觉上说，注释不但不是一种坏味道，事实上它们还是一种香味呢。我们之所以要在这里提到注释，是因为人们常把它当作“除臭剂”来使用。常常会有这样的情况：你看到一段代码有着长长的注释，然后发现，这些注释之所以存在乃是因为代码很糟糕。这种情况的发生次数之多，实在令人吃惊。

注释可以带我们找到本章先前提到的各种坏味道。找到坏味道后，我们首先应该以各种重构手法把坏味道去除。完成之后我们常常会发现：注释已经变得多余了，因为代码已经清楚地说明了一切。

如果你需要注释来解释一块代码做了什么，试试提炼函数（106）；如果函数已经提炼出来，但还是需要注释来解释其行为，试试用改变函数声明（124）为它改名；如果你需要注释说明某些系统的需求规格，试试引入断言（302）。

当你感觉需要撰写注释时，请先尝试重构，试着让所有注释都变得多余。

如果你不知道该做什么，这才是注释的良好运用时机。除了用来记述将来的打算之外，注释还可以用来标记你并无十足把握的区域。你可以在注释里写下自己“为什么做某某事”。这类信息可以帮助将来的修改者，尤其是那些健忘的家伙。

第4章　构筑测试体系

重构是很有价值的工具，但只有重构还不行。要正确地进行重构，前提是得有一套稳固的测试集合，以帮我发现难以避免的疏漏。即便有工具可以帮我自动完成一些重构，很多重构手法依然需要通过测试集合来保障。

我并不把这视为缺点。我发现，编写优良的测试程序，可以极大提高我的编程速度，即使不进行重构也一样如此。这让我很吃惊，也违反许多程序员的直觉，所以我有必要解释一下这个现象。

4.1 自测试代码的价值

如果你认真观察大多数程序员如何分配他们的时间，就会发现，他们编写代码的时间仅占所有时间中很少的一部分。有些时间用来决定下一步干什么，有些时间花在设计上，但是，花费在调试上的时间是最多的。我敢肯定，每一位读者一定都记得自己花数小时调试代码的经历——而且常常是通宵达旦。每个程序员都能讲出一个为了修复一个bug花费了一整天（甚至更长时间）的故事。修复bug通常是比较快的，但找出bug所在却是一场噩梦。当修复一个bug时，常常会引起另一个bug，却在很久之后才会注意到它。那时，你又要花上大把时间去定位问题。

我走上“自测试代码”这条路，源于1992年OOPSLA大会上的一个演讲。有个人（我记得好像是Bedarra公司的Dave Thomas）提到：“类应该包含它们自己的测试代码。”这让我决定，将测试代码和产品代码一起放到代码库中。

当时，我正在迭代方式开发一个软件，因此，我尝试在每个迭代结束后把测试代码加上。当时我的软件项目很小，我们每周进行一次迭代。所以，

运行测试变得相当简单——尽管非常简单，但也非常枯燥。因为每个测试都把测试结果输出到控制台中，我必须逐一检查它们。我是一个很懒的人，所以总是在当下努力工作，以免日后有更多的活儿。我意识到，其实完全不必自己盯着屏幕检验测试输出的信息是否正确，而是让计算机来帮我做检查。我需要做的就是把我所期望的输出放到测试代码中，然后做一个对比就行了。于是，我只要运行所有测试用例，假如一切都没问题，屏幕上就只出现一个“OK”。现在我的代码都能够“自测试”了。

从此，运行测试就像执行编译一样简单。于是，我每次编译时都会运行测试。不久之后，我注意到自己的开发效率大大提高。我意识到，这是因为我没有花太多时间去测试的缘故。如果我不小心引入一个可被现有测试捕捉到的bug，那么只要运行测试，它就会向我报告这个bug。由于代码原本是可以正常运行的，所以我知道这个bug必定是在前一次运行测试后修改代码引入的。由于我频繁地运行测试，每次测试都在不久之前，因此我知道bug的源头就是我刚刚写下的代码。因为代码量很少，我对它也记忆犹新，所以就能轻松找出bug。从前需要一小时甚至更多时间才能找到的bug，现在最多只要几分钟就找到了。之所以能够拥有如此强大的bug侦测能力，不仅仅是因为我的代码能够自测试，也得益于我频繁地运行它们。

💡 确保所有测试都完全自动化，让它们检查自己的测试结果。

注意到这一点后，我对测试的积极性更高了。我不再等待每次迭代结尾时再增加测试，而是只要写好一点功能，就立即添加它们。每天我都会添加一些新功能，同时也添加相应的测试。这样，我很少花超过几分钟的时间来追查回归错误。

从我最早的试验开始到现在为止，编写和组织自动化测试的工具已经有了长足的发展。1997年，Kent Beck从瑞士飞往亚特兰大去参加当年的OOPSLA会议，在飞机上他与Erich Gamma结对，把他为Smalltalk撰写的测试框架移植到了Java上。由此诞生的JUnit框架在测试领域影响力非凡，也在不同的编程语言中催生了很多类似的工具[mf-xunit]。

💡 一套测试就是一个强大的bug侦测器，能够大大缩减查找bug所需的时间。

我得承认，说服别人也这么做并不容易。编写测试程序，意味着要写很多额外的代码。除非你确实体会到这种方法是如何提升编程速度的，否则自

测试似乎就没什么意义。很多人根本没学过如何编写测试程序，甚至根本没考虑过测试，这对于编写自测试也很不利。如果测试需要手动运行，那的确是令人烦闷。但是，如果测试可以自动运行，编写测试代码就会真的很有趣。

事实上，撰写测试代码的最好时机是在开始动手编码之前。当我需要添加特性时，我会先编写相应的测试代码。听起来离经叛道，其实不然。编写测试代码其实就是在问自己：为了添加这个功能，我需要实现些什么？编写测试代码还能帮我把注意力集中于接口而非实现（这永远是一件好事）。预先写好的测试代码也为我的工作安上一个明确的结束标志：一旦测试代码正常运行，工作就可以结束了。

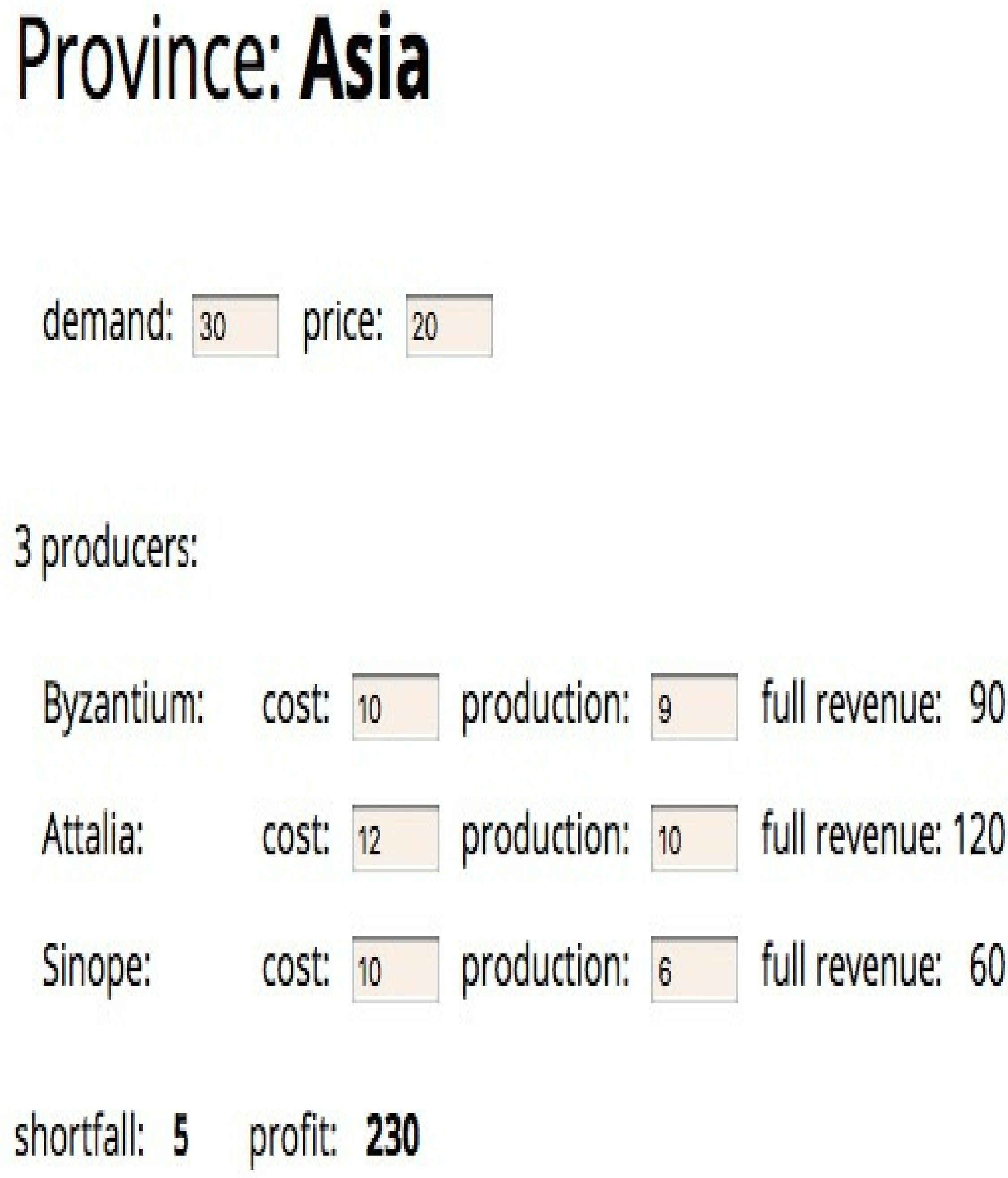
Kent Beck将这种先写测试的习惯提炼成一门技艺，叫测试驱动开发（Test-Driven Development，TDD）[mf-tdd]。测试驱动开发的编程方式依赖于下面这个短循环：先编写一个（失败的）测试，编写代码使测试通过，然后进行重构以保证代码整洁。这个“测试、编码、重构”的循环应该在每个小时内都完成很多次。这种良好的节奏感可使编程工作以更加高效、有条不紊的方式开展。我就不在这里再做更深入的介绍，但我自己确实经常使用，也非常建议你试一试。

大道理先放在一边。尽管我相信每个人都可以从编写自测试代码中收益，但这并不是本书的重点。本书谈的是重构，而重构需要测试。如果你想重构，就必须编写测试。本章会带你入门，教你如何在JavaScript中编写简单的测试，但它不是一本专门讲测试的书，所以我不想讲得太细。但我发现，少许测试往往就足以带来惊人的收益。

和本书其他内容一样，我以示例来介绍测试手法。开发软件的时候，我一边写代码，一边写测试。但有时我也需要重构一些没有测试的代码。在重构之前，我得先改造这些代码，使其能够自测试才行。

4.2 待测试的示例代码

这里我展示了一份有待测试的代码。这份代码来自一个简单的应用，用于支持用户查看并调整生产计划。它的（略显粗糙的）界面长得像下面这张图所示的这样。



每个行省（province）都有一份生产计划，计划中包含需求量（demand）和采购价格（price）。每个行省都有一些生产商（producer），他们各自以不同的成本价（cost）供应一定数量的产品。界

面上还会显示，当商家售出所有的商品时，他们可以获得的总收入（full revenue）。页面底部展示了该区域的产品缺额（需求量减去总产量）和总利润（profit）。用户可以在界面上修改需求量及采购价格，以及不同生产商的产量（production）和成本价，以观察缺额和总利润的变化。用户在界面上修改任何数值时，其他的数值都会同时得到更新。

这里我展示了一个用户界面，是为了让你了解该应用的使用方式，但我只会聚焦于软件的业务逻辑部分，也就是那些计算利润和缺额的类，而非那些生成HTML或监听页面字段更新的代码。本章只是先带你走进自测试代码世界的大门，因而最好是从最简单的例子开始，也就是那些不涉及用户界面、持久化或外部服务交互的代码。这种隔离的思路其实在任何场景下都适用：一旦业务逻辑的部分开始变复杂，我就会把它与UI分离开，以便能更好地理解和测试它。

这块业务逻辑代码涉及两个类：一个代表了单个生产商（`Producer`），另一个用来描述一个行省（`Province`）。`Province`类的构造函数接收一个JavaScript对象，这个对象的内容我们可以想象是由一个JSON文件提供的。

下面的代码能从JSON文件中构造出一个行省对象。

class Province...

```
constructor(doc) {
  this._name = doc.name;
  this._producers = [];
  this._totalProduction = 0;
  this._demand = doc.demand;
  this._price = doc.price;

doc.producers.forEach(d => this.addProducer(new Producer(this
}
addProducer(arg) {
  this._producers.push(arg);
  this._totalProduction += arg.production;
}
```

下面的函数会创建可用的JSON数据，我可以用它的返回值来构造一个行省对象，并拿这个对象来做测试。

顶层作用域...

```
function sampleProvinceData() {
  return {
    name: "Asia",
    producers: [
      {name: "Byzantium", cost: 10, production: 9},
      {name: "Attalia",   cost: 12, production: 10},
      {name: "Sinope",    cost: 10, production: 6},
    ],
    demand: 30,
    price: 20
  };
}
```

行省类中有许多设值函数和取值函数，它们用于获取各类数据的值。

class Province...

```
get name()    {return this._name;}
get producers() {return this._producers.slice();}
get totalProduction()    {return this._totalProduction;}
set totalProduction(arg) {this._totalProduction = arg;}
get demand()    {return this._demand;}
set demand(arg) {this._demand = parseInt(arg);}
get price()    {return this._price;}
set price(arg) {this._price = parseInt(arg);}
```

设值函数会被UI端调用，接收一个包含数值的字符串。我需要将它们转换成数值，以便在后续的计算中使用。

代表生产商的`Producer`类则基本只是一个存放数据的容器。

class Producer...

```
constructor(aProvince, data) {
  this._province = aProvince;
  this._cost = data.cost;
  this._name = data.name;
  this._production = data.production || 0;
}
get name() {return this._name;}
```

```
get cost() {return this._cost;}
set cost(arg) {this._cost = parseInt(arg);}

get production() {return this._production;}
set production(amountStr) {
  const amount = parseInt(amountStr);
  const newProduction = Number.isNaN(amount) ? 0 : amount;

this._province.totalProduction += newProduction - this._produ
  this._production = newProduction;
}
```

在设值函数production中更新派生数据的方式有点丑陋，每当看到这种代码，我便想通过重构帮它改头换面。但在重构之前，我必须记得先为它添加测试。

缺额的计算逻辑也很简单。

class Province...

```
get shortfall() {
  return this._demand - this.totalProduction;
}
```

计算利润的逻辑则要相对复杂一些。

class Province...

```
get profit() {
  return this.demandValue - this.demandCost;
}
get demandCost() {
  let remainingDemand = this.demand;
  let result = 0;
  this.producers
    .sort((a,b) => a.cost - b.cost)
    .forEach(p => {

const contribution = Math.min(remainingDemand, p.production);
       remainingDemand -= contribution;
       result += contribution * p.cost;
    });
  return result;
}
get demandValue() {
  return this.satisfiedDemand * this.price;
}
get satisfiedDemand() {
  return Math.min(this._demand, this.totalProduction);
}
```

4.3　第一个测试

开始测试这份代码前，我需要一个测试框架。JavaScript世界里这样的框架有很多，这里我选用的是使用度和声誉都还不错的Mocha。我不打算全面讲解框架的使用，而只会用它写一些测试作为例子。看完之后，你应该能轻松地学会用别的框架来编写类似的测试。

以下是为缺额计算过程编写的一个简单的测试：

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    assert.equal(asia.shortfall, 5);
  });
});
```

Mocha框架组织测试代码的方式是将其分组，每一组下包含一套相关的测试。测试需要写在一个it块中。对于这个简单的例子，测试包含了两个步骤。第一步设置好一些测试夹具（fixture），也就是测试所需要的数据和对象等（就本例而言是一

个加载好了的行省对象）；第二步则是验证测试夹具是否具备某些特征（就本例而言则是验证算出的缺额应该是期望的值）。

> 不同开发者在describe和it块里撰写的描述信息各有不同。有的人会写一个描述性的句子解释测试的内容，也有人什么都不写，认为所谓描述性的句子跟注释一样，不外乎是重复代码已经表达的东西。我个人不喜欢多写，只要测试失败时足以识别出对应的测试就够了。

如果我在NodeJS的控制台下运行这个测试，那么其输出看起来是这样：

```
''''''''''''''

  1 passing (61ms)
```

它的反馈极其简洁，只包含了已运行的测试数量以及测试通过的数量。

当我为类似的既有代码编写测试时，发现一切正常工作固然是好，但我天然持怀疑精神。特别是有很多测试在运行时，我总会担心测试没有按我期望的方式检查结果，从而没法在实际出错的时候抓

到bug。因此编写测试时，我想看到每个测试都至少失败一遍。我最爱的方式莫过于在代码中暂时引入一个错误，像这样：

总是确保测试不该通过时真的会失败。

class Province...

```
get shortfall() {
  return this._demand - this.totalProduction * 2;
}
```

现在控制台的输出就有所改变了：

```
!

  0 passing (72ms)
  1 failing

1) province shortfall:
   AssertionError: expected -20 to equal 5
    at Context.<anonymous> (src/tester.js:10:12)
```

框架会报告哪个测试失败了，并给出失败的根本原因——这里是因为实际算出的值与期望的值不

相符。于是我总算见到有什么东西失败了，并且还能马上看到是哪个测试失败，获得一些出错的线索（这个例子中，我还能确认这就是我引入的那个错误）。

一个真实的系统可能拥有数千个测试。好的测试框架应该能帮我简单快速地运行这些测试，一旦出错，我能马上看到。尽管这种反馈非常简单，但对自测试代码来说却尤为重要。工作时我会非常频繁地运行测试，要么是检验新代码的进展，要么是检查重构过程是否出错。

频繁地运行测试。对于你正在处理的代码，与其对应的测试至少每隔几分钟就要运行一次，每天至少运行一次所有的测试。

Mocha框架允许使用不同的库（它称之为断言库）来验证测试的正确性。JavaScript世界的断言库，连在一起都可以绕地球一周了，当你读到这里时，可能有些仍然还没过时。我现在使用的库是Chai，它可以支持我编写不同类型的断言，比如“assert”风格的：

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
```

```
    assert.equal(asia.shortfall, 5);
  });
});
```

或者是“expect”风格的：

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.shortfall).equal(5);
  });
});
```

一般来讲我更倾向于使用assert风格的断言，但使用JavaScript时我倒是更常使用expect的风格。

环境不同，运行测试的方式也不同。使用Java编程时，我使用IDE的图形化测试运行界面。它有一个进度条，所有测试都通过时就会显示绿色；只要有任何测试失败，它就会变成红色。我的同事们经常使用“绿色条”和“红色条”来指代测试的状态。我可能会讲“看到红条时永远不许进行重构”，意思是：测试集合中还有失败的测试时就不应该先去重构。有时我也会讲“回退到绿条”，表示你应该撤销最近一次更改，将测试恢复到上一次全部通过的状态（通常是切回到版本控制的最近一次提交点）。

图形化测试界面的确很棒，但并不是必需的。我通常会在Emacs中配置一个运行测试的快捷键，

然后在编译窗口中观察纯文本的反馈。要点在于，我必须能快速地知道测试是否全部都通过了。

4.4　再添加一个测试

现在，我将继续添加更多测试。我遵循的风格是：观察被测试类应该做的所有事情，然后对这个类的每个行为进行测试，包括各种可能使它发生异常的边界条件。这不同于某些程序员提倡的“测试所有`public`函数”的风格。记住，测试应该是一种风险驱动的行为，我测试的目标是希望找出现在或未来可能出现的bug。所以我不会去测试那些仅仅读或写一个字段的访问函数，因为它们太简单了，不太可能出错。

这一点很重要，因为如果尝试撰写过多测试，结果往往反而导致测试不充分。事实上，即使我只做一点点测试，也从中获益良多。测试的重点应该是那些我最担心出错的部分，这样就能从测试工作中得到最大利益。

接下来，我的目光落到了代码的另一个主要输出上，也就是总利润的计算。我同样可以在一开始的测试夹具上，对总利润做一个基本的测试。

编写未臻完善的测试并经常运行，好

过对完美测试的无尽等待。

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.shortfall).equal(5);
  });
  it('profit', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.profit).equal(230);
  });
});
```

这是最终写出来的测试，但我是怎么写出它来的呢？首先我随便给测试的期望值写了一个数，然后运行测试，将程序产生的实际值（230）填回去。当然，我也可以自己手动计算，不过，既然现在的代码是能正常运行的，我就选择暂时相信它。测试可以正常工作后，我又故技重施，在利润的计算过程插入一个假的乘以2逻辑来破坏测试。如我所料，测试会失败，这时我才满意地将插入的假逻辑恢复过来。这个模式是我为既有代码添加测试时最常用的方法：先随便填写一个期望值，再用程序产生的真实值来替换它，然后引入一个错误，最后恢复错误。

这个测试随即产生了一些重复代码——它们都在第一行里初始化了同一个测试夹具。正如我对一般的重复代码抱持怀疑，测试代码中的重复同样令我心生疑惑，因此我要试着将它们提到一处公共的

地方，以此来消灭重复。一种方案就是把常量提取到外层作用域里。

```
describe('province', function() {

const asia = new Province(sampleProvinceData()); // DON'T DO
  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });
  it('profit', function() {
    expect(asia.profit).equal(230);
  });
});
```

但正如代码注释所说的，我从不这样做。这样做的确能解决一时的问题，但共享测试夹具会使测试间产生交互，这是滋生bug的温床——还是你写测试时能遇见的最恶心的bug之一。使用了JavaScript中的const关键字只表明asia的引用不可修改，不表明对象的内容也不可修改。如果未来有一个测试改变了这个共享对象，测试就可能时不时失败，因为测试之间会通过共享夹具产生交互，而测试的结果就会受测试运行次序的影响。测试结果的这种不确定性，往往使你陷入漫长而又艰难的调试，严重时甚至可能令你对测试体系的信心产生动摇。因此，我比较推荐采取下面的做法：

```
describe('province', function() {
  let asia;
  beforeEach(function() {
    asia = new Province(sampleProvinceData());
```

```
  });
  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });
  it('profit', function() {
    expect(asia.profit).equal(230);
  });
});
```

beforeEach子句会在每个测试之前运行一遍，将asia变量清空，每次都给它赋一个新的值。这样我就能在每个测试开始前，为它们各自构建一套新的测试夹具，这保证了测试的独立性，避免了可能带来麻烦的不确定性。

对于这样的建议，有人可能会担心，每次创建一个崭新的测试夹具会拖慢测试的运行速度。大多数时候，时间上的差别几乎无法察觉。如果运行速度真的成为问题，我也可以考虑共享测试夹具，但这样我就得非常小心，确保没有测试会去更改它。如果我能够确定测试夹具是百分之百不可变的，那么也可以共享它。但我的本能反应还是要使用独立的测试夹具，可能因为我过去尝过了太多共享测试夹具带来的苦果。

既然我在beforeEach里运行的代码会对每个测试生效，那么为何不直接把它挪到每个it块里呢？让所有测试共享一段测试夹具代码的原因，是为了使我对公用的夹具代码感到熟悉，从而将眼光聚焦

于每个测试的不同之处。beforeEach块旨在告诉读者，我使用了同一套标准夹具。你可以接着阅读describe块里的所有测试，并知道它们都是基于同样的数据展开测试的。

4.5　修改测试夹具

加载完测试夹具后，我编写了一些测试来探查它的一些特性。但在实际应用中，该夹具可能会被频繁更新，因为用户可能在界面上修改数值。

大多数更新都是通过设值函数完成的，我一般也不会测试这些方法，因为它们不太可能出什么bug。不过Producer类中的产量（production）字段，其设值函数行为比较复杂，我觉得它倒是值得一测。

describe('province'...

```
it('change production', function() {
    asia.producers[0].production = 20;
    expect(asia.shortfall).equal(-6);
    expect(asia.profit).equal(292);
});
```

这是一个常见的测试模式。我拿到beforeEach配置好的初始标准夹具，然后对该夹具进行必要的检查，最后验证它是否表现出我期望的行为。如果你读过测试相关的资料，就会经常听到各种类似的

术语，比如配置-检查-验证（setup-exercise-verify）、given-when-then或者准备-行为-断言（arrange-act-assert）等。有时你能在一个测试里见到所有的步骤，有时那些早期的公用阶段会被提到一些标准的配置步骤里，诸如`beforeEach`等。

> （其实还有第四个阶段，只是不那么明显，一般很少提及，那就是拆除阶段。此阶段可将测试夹具移除，以确保不同测试之间不会产生交互。因为我是在`beforeEach`中配置好数据的，所以测试框架会默认在不同的测试间将我的测试夹具移除，相当于我自动享受了拆除阶段带来的便利。多数测试文献的作者对拆除阶段一笔带过，这可以理解，因为多数时候我们可以忽略它。但有时因为创建缓慢等原因，我们会在不同的测试间共享测试夹具，此时，显式地声明一个拆除操作就是很重要的。）

在这个测试中，我在一个`it`语句里验证了两个不同的特性。作为一个基本规则，一个`it`语句中最好只有一个验证语句，否则测试可能在进行第一个验证时就失败，这通常会掩盖一些重要的错误信息，不利于你了解测试失败的原因。不过，在上面的场景中，我觉得两个断言本身关系非常紧密，写

在同一个测试中问题不大。如果稍后需要将它们分离到不同的`it`语句中，我可以到时再做。

4.6 探测边界条件

到目前为止我的测试都聚焦于正常的行为上，这通常也被称为“正常路径”（happy path），它指的是一切工作正常、用户使用方式也最符合规范的那种场景。同时，把测试推到这些条件的边界处也是不错的实践，这可以检查操作出错时软件的表现。

无论何时，当我拿到一个集合（比如说此例中的生产商集合）时，我总想看看集合为空时会发生什么。

```
describe('no producers', function() {
  let noProducers;
  beforeEach(function() {
    const data = {
      name: "No proudcers",
      producers: [],
      demand: 30,
      price: 20
    };
    noProducers = new Province(data);
  });
  it('shortfall', function() {
    expect(noProducers.shortfall).equal(30);
  });
  it('profit', function() {
    expect(noProducers.profit).equal(0);
  });
```

如果拿到的是数值类型，0会是不错的边界条件：

describe('province'...

```
it('zero demand', function() {
  asia.demand = 0;
    expect(asia.shortfall).equal(-25);
    expect(asia.profit).equal(0);
  });
```

负值同样值得一试：

describe('province'...

```
it('negative demand', function() {
  asia.demand = -1;
  expect(asia.shortfall).equal(-26);
  expect(asia.profit).equal(-10);
});
```

测试到这里，我不禁有一个想法：对于这个业务领域来讲，提供一个负的需求值，并算出一个负的利润值意义何在？最小的需求量不应该是0吗？或许，设值方法需要对负值有些不同的行为，比如抛出错误，或总是将值设置为0。这些问题都很好，编写这样的测试能帮助我思考代码本应如何应

对边界场景。

设值函数接收的字符串是从UI上的字段读来的，它已经被限制为只能填入数字，但仍然有可能是空字符串，因此同样需要测试来保证代码对空字符串的处理方式符合我的期望。

考虑可能出错的边界条件，把测试火力集中在那儿。

describe('province'...

```
it('empty string demand', function() {
  asia.demand = "";
  expect(asia.shortfall).NaN;
  expect(asia.profit).NaN;
});
```

可以看到，我在这里扮演“程序公敌”的角色。我积极思考如何破坏代码。我发现这种思维能够提高生产力，并且很有趣——它纵容了我内心中比较促狭的那一部分。

这个测试结果很有意思：

```
describe('string for producers', function() {
```

```
it('', function() {
  const data = {
    name: "String producers",
    producers: "",
    demand: 30,
    price: 20
  };
  const prov = new Province(data);
  expect(prov.shortfall).equal(0);
});
```

它并不是抛出一个简单的错误说缺额的值不为0。控制台的报错输出实际如下：

```
'''''''''!

  9 passing (74ms)
  1 failing

  1) string for producers :
     TypeError: doc.producers.forEach is not a function
       at new Province (src/main.js:22:19)
       at Context.<anonymous> (src/tester.js:86:18)
```

Mocha把这也当作测试失败（failure），但多数测试框架会把它当作一个错误（error），并与正常的测试失败区分开。“失败”指的是在验证阶段中，实际值与验证语句提供的期望值不相等；而这里的“错误”则是另一码事，它是在更早的阶段前抛出的异常（这里是在配置阶段）。它更像代码的作者没有预料到的一种异常场景，因此我们不幸地得到了每个JavaScript程序员都很熟悉的错误（“...is

not a function”）。

那么代码应该如何处理这种场景呢？一种思路是，对错误进行处理并给出更好的出错响应，比如说抛出更有意义的错误信息，或是直接将producers字段设置为一个空数组（最好还能再记录一行日志信息）。但维持现状不做处理也说得通，也许该输入对象是由可信的数据源提供的，比如同个代码库的另一部分。在同一代码库的不同模块之间加入太多的检查往往会导致重复的验证代码，它带来的好处通常不抵害处，特别是你添加的验证可能在其他地方早已做过。但如果该输入对象是由一个外部服务所提供，比如一个返回JSON数据的请求，那么校验和测试就显得必要了。不论如何，为边界条件添加测试总能引发这样的思考。

如果这样的测试是在重构前写出的，那么我很可能还会删掉它。重构应该保证可观测的行为不发生改变，而类似的错误已经超越可观测的范畴。删掉这条测试，我就不用担心重构过程改变了代码对这个边界条件的处理方式。

> 如果这个错误会导致脏数据在应用中到处传递，或是产生一些很难调试的失败，我可能会用引入断言（302）手法，使代码不满足预设

条件时快速失败。我不会为这样的失败断言添加测试，它们本身就是一种测试的形式。

什么时候应该停下来？我相信这样的话你已经听过很多次："任何测试都不能证明一个程序没有bug。"确实如此，但这并不影响"测试可以提高编程速度"。我曾经见过好几种测试规则建议，其目的都是保证你能够测试所有情况的一切组合。这些东西值得一看，但是别让它们影响你。当测试数量达到一定程度之后，继续增加测试带来的边际效用会递减；如果试图编写太多测试，你也可能因为工作量太大而气馁，最后什么都写不成。你应该把测试集中在可能出错的地方。观察代码，看哪儿变得复杂；观察函数，思考哪些地方可能出错。是的，你的测试不可能找出所有bug，但一旦进行重构，你可以更好地理解整个程序，从而找到更多bug。虽然在开始重构之前我会确保有一个测试套件存在，但前进途中我总会加入更多测试。

不要因为测试无法捕捉所有的bug就不写测试，因为测试的确可以捕捉到大多数bug。

4.7　测试远不止如此

本章我想讨论的东西到这里就差不多了，毕竟这是一本关于重构而不是测试的书。但测试本身是一个很重要的话题，它既是重构所必要的基础保障，本身也是一个有价值的工具。自本书第1版以来，我很高兴看到重构作为一项编程实践在逐步发展，但我更高兴见到业界对测试的态度也在发生转变。之前，测试更多被认为是另一个独立的（所需专业技能也较少的）团队的责任，但现在它愈发成为任何一个软件开发者所必备的技能。如今一个架构的好坏，很大程度要取决于它的可测试性，这是一个好的行业趋势。

这里我展示的测试都属于单元测试，它们负责测试一块小的代码单元，运行足够快速。它们是自测试代码的支柱，是一个系统中占绝大多数的测试类型。同时也有其他种类的测试存在，有的专注于组件之间的集成，有的会检验软件跨越几个层级的运行结果，有的用于查找性能问题，不一而足。（而且，同行们对于如何归类测试的争论，恐怕比繁多的测试种类本身还要多。）

与编程的许多方面类似，测试也是一种迭代式

的活动。除非你技能非常纯熟，或者非常幸运，否则你很难第一次就把测试写对。我发觉我持续地在测试集上工作，就与我在主代码库上的工作一样多。很自然，这意味着我在增加新特性时也要同时添加测试，有时还需要回顾已有的测试：它们足够清晰吗？我需要重构它们，以帮助我更好地理解吗？我拥有的测试是有价值的吗？一个值得养成的好习惯是，每当你遇见一个bug，先写一个测试来清楚地复现它。仅当测试通过时，才视为bug修完。只要测试存在一天，我就知道这个错误永远不会再复现。这个bug和对应的测试也会提醒我思考：测试集里是否还有这样不被阳光照耀到的犄角旮旯？

一个常见的问题是，“要写多少测试才算足够？”这个问题没有很好的衡量标准。有些人拥护以测试覆盖率[mf-tc]作为指标，但测试覆盖率的分析只能识别出那些未被测试覆盖到的代码，而不能用来衡量一个测试集的质量高低。

每当你收到bug报告，请先写一个单元测试来暴露这个bug。

一个测试集是否足够好，最好的衡量标准其实

是主观的，请你试问自己：如果有人在代码里引入了一个缺陷，你有多大的自信它能被测试集揪出来？这种信心难以被定量分析，盲目自信不应该被计算在内，但自测试代码的全部目标，就是要帮你获得此种信心。如果我重构完代码，看见全部变绿的测试就可以十分自信没有引入额外的bug，这样，我就可以高兴地说，我已经有了一套足够好的测试。

测试同样可能过犹不及。测试写得太多的一个征兆是，相比要改的代码，我在改动测试上花费了更多的时间——并且我能感到测试就在拖慢我。不过尽管过度测试时有发生，相比测试不足的情况还是稀少得多。

第5章　介绍重构名录

本书剩余的篇幅是一份重构的名录。最初这个名录只是我的个人笔记，我用它来提示自己如何以安全且高效的方式进行重构。然后我不断精炼这份名录，对一些重构的深入探索又引出了更多的重构手法。对于不太常用的重构手法，我还是会不断参阅这份名录。

5.1　重构的记录格式

介绍重构时，我采用一种标准格式。每个重构手法都有如下5个部分。

- 首先是名称（name）。要建造一个重构词汇表，名称是很重要的。这个名称也就是我将在本书其他地方使用的名称。如今重构经常会有多个名字，所以我会同时列出常见的别名。
- 名称之后是一个简单的速写（sketch）。这部分可以帮助你更快找到你所需要的重构手法。
- 动机（motivation）为你介绍“为什么需要做这个重构”和“什么情况下不该做这个重构”。
- 做法（mechanics）简明扼要地一步一步介绍如何进行此重构。
- 范例（examples）以一个十分简单的例子说明此重构手法如何运作。

速写部分会以代码示例的形式展示重构带来的转变。速写的用意不是解释重构的用途，更不是详细讲解如何操作这个重构；但如果你曾经看过这个重构手法，速写能帮你回忆起它。如果你是第一次接触到这个重构手法，可能最好是先阅读范例部分。我还给每个重构手法画了一幅小图。同样，我

也不指望这些小图能说清重构手法的内容，只是提供一点图像记忆的线索。

“做法”出自我自己的笔记。这些笔记是为了让我在一段时间不做某项重构之后还能记得怎么做。它们也颇为简洁，通常不会解释“为什么要这么做那么做”。我会在“范例”中给出更多解释。这么一来，“做法”就成了简短的笔记。如果你知道该使用哪个重构，但记不清具体步骤，可以参考“做法”部分（至少我是这么使用它们的）；如果你初次使用某个重构，可能只参考“做法”还不够，你还需要阅读“范例”。

撰写“做法”的时候，我尽量将重构的每个步骤拆得尽可能小。我强调安全的重构方式，所以应该采用非常小的步骤，并且在每个步骤之后进行测试。真正工作时，我通常会采用比这里介绍的“婴儿学步”稍大些的步骤，然而一旦出问题，我就会撤销上一步，换用比较小的步骤。这些步骤还包含一些特殊情况的参考，所以它们也有检查清单的作用。我自己经常忘掉这些该做的事情。

绝大多数时候我只列出了重构的一套做法，但其实一个重构并非只有一套做法。我在本书中选择介绍这些做法，因为它们大多数时候都管用。等你经过练习获得更多重构经验，你可能会调整重构的做法，那完全没问题。只要牢记一点：小步前进，

情况越复杂，步子就要越小。

“范例”像是简单而有趣的教科书。我使用这些范例是为了帮助解释重构的基本要素，最大限度地避免其他枝节，所以我希望你能原谅其中的简化工作（它们当然不是优秀的业务建模例子）。不过我敢肯定，你一定能在那些更复杂的情况中使用它们。某些十分简单的重构干脆没有范例，因为我觉得为它们加上一个范例不会有多大意义。

更明确地说，加上范例仅仅是为了阐释当时讨论的重构手法。通常那些代码最终仍有其他问题，但修正那些问题需要用到其他重构手法。某些情况下数个重构经常被一并运用，这时候我会把范例带到另一个重构中继续使用。大部分时候，一个范例只为一项重构而设计，这么做是为了让每一项重构手法自成一体，因为这份重构名录的首要目的还是作为参考工具。

修改后的代码可能被埋没在未修改的代码中，难以一眼看出，所以我使用不同的颜色突出显示修改过的代码。但我并没有突出显示所有修改过的代码，因为一旦修改过的代码太多，全都突出显示反而不能突显重点。

5.2　挑选重构的依据

这不是一份巨细靡遗的重构名录。我只是认为这些重构手法最值得被记录下来。之所以说它们“最值得”，因为这些都是很常用的重构手法，并且值得给它们命名和详细的介绍：其中一些做法很有意思，能帮助读者提高整体重构技能水平，另外一些则对于代码设计质量的提升效果显著。

有些重构没有进入这份名录，因为它们太小、太简单，我觉得没必要多加赘述。例如，在撰写第1版时我就曾经考虑过移动语句（223），这个重构我经常使用，但我觉得没必要将它放进名录里（显然我在写第2版的时候改变了想法）。以后也许还有类似这样的重构会被加进书里，不过那要看我投入多少精力在新增重构上了。

还有一些没有进入名录的重构，要么是我用得很少，要么是与其他重构非常相似。本书中的每个重构，逻辑上来说，都有一个反向的重构。但我并没有把所有反向重构都写下来，因为我发现很多反向重构没太大意思。例如，封装变量（132）是一个常用又好用的重构，但它的反向重构我几乎从来不会做（而且就算要做也非常简单），所以我觉得

没必要将这个反向重构放进名录。

第6章　第一组重构

在重构名录的开头，我首先介绍一组我认为最有用的重构。

我最常用到的重构就是用提炼函数（106）将代码提炼到函数中，或者用提炼变量（119）来提炼变量。既然重构的作用就是应对变化，你应该不会感到惊讶，我也经常使用这两个重构的反向重构——内联函数（115）和内联变量（123）。

提炼的关键就在于命名，随着理解的加深，我经常需要改名。改变函数声明（124）可以用于修改函数的名字，也可以用于添加或删减参数。变量也可以用变量改名（137）来改名，不过需要先做封装变量（132）。在给函数的形式参数改名时，不妨先用引入参数对象（140）把常在一起出没的参数组合成一个对象。

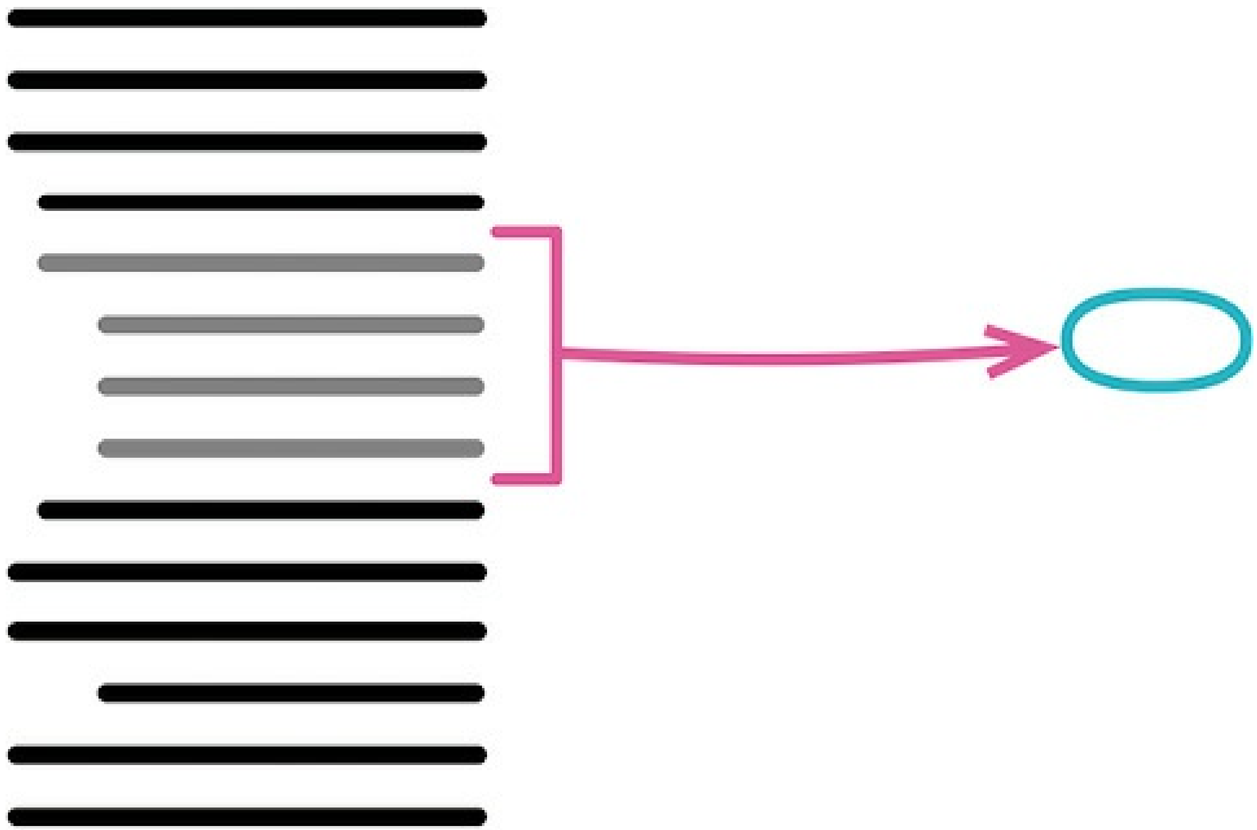
形成函数并给函数命名，这是低层级重构的精髓。有了函数以后，就需要把它们组合成更高层级的模块。我会使用函数组合成类（144），把函数和它们操作的数据一起组合成类。另一条路径是用

函数组合成变换（149）将函数组合成变换式（transform），这对于处理只读数据尤为便利。再往前一步，常常可以用拆分阶段（154）将这些模块组成界限分明的处理阶段。

6.1　提炼函数（Extract Function）

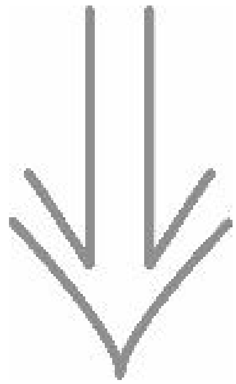
曾用名：提炼函数（Extract Method）

反向重构：内联函数（115）



```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding();

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
}
```



```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding();
  printDetails(outstanding);

  function printDetails(outstanding) {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
  }
}
```

动机

提炼函数是我最常用的重构之一。（在这儿我用了“函数/function”这个词，但换成面向对象语言中的“方法/method”，或者其他任何形式的“过程/procedure”或者“子程序/subroutine”，也同样适用。）我会浏览一段代码，理解其作用，然后将其提炼到一个独立的函数中，并以这段代码的用途为这个函数命名。

对于“何时应该把代码放进独立的函数”这个问题，我曾经听过多种不同的意见。有的观点从代码的长度考虑，认为一个函数应该能在一屏中显示。有的观点从复用的角度考虑，认为只要被用过不止一次的代码，就应该单独放进一个函数；只用过一次的代码则保持内联（inline）的状态。但我认为最合理的观点是“将意图与实现分开”：如果你需要花时间浏览一段代码才能弄清它到底在干什么，那么就应该将其提炼到一个函数中，并根据它所做的事为其命名。以后再读到这段代码时，你一眼就能看到函数的用途，大多数时候根本不需要关心函数如何达成其用途（这是函数体内干的事）。

一旦接受了这个原则，我就逐渐养成一个习惯：写非常小的函数——通常只有几行的长度。在我看来，一个函数一旦超过6行，就开始散发臭味。我甚至经常会写一些只有1行代码的函数。Kent Beck曾向我展示最初的Smalltalk系统中的一个例子，从那时起我就接受了“函数名的长度不重要”的观念。那时运行Smalltalk的计算机只有黑白屏显示器，如果你想高亮突显某些文本或图像，就需要反转视频的显示。为此，Smalltalk用于控制图像显示的类有一个叫作`highlight`的方法，其中的实现就只是调用`reverse`方法。在这个例子里，`highlight`方法的名字比实现还长，但这并不重要，因为在这个方法中，代码的意图与实现之间

有着相当大的距离。

有些人担心短函数会造成大量函数调用，因而影响性能。在我尚且年轻时，有时确实会有这个问题；但如今“由于函数调用影响性能”的情况已经非常罕见了。短函数常常能让编译器的优化功能运转更良好，因为短函数可以更容易地被缓存。所以，应该始终遵循性能优化的一般指导方针，不用过早担心性能问题。

小函数得有个好名字才行，所以你必须在命名上花心思。起好名字需要练习，不过一旦你掌握了其中的技巧，就能写出很有自描述性的代码。

我经常会看见这样的情况：在一个大函数中，一段代码的顶上放着一句注释，说明这段代码要做什么。在把这段代码提炼到自己的函数中时，这样的注释往往会提示一个好名字。

做法

- 创造一个新函数，根据这个函数的意图来对它命名（以它“做什么”来命名，而不是以它“怎样做”命名）。

如果想要提炼的代码非常简单，例如只是一个函数调用，只要新函数的名称能够以更好的方式昭示代码意图，我还是会提炼它；但如果想不出一个更有意义的名称，这就是一个信号，可能我不应该提炼这块代码。不过，我不一定非得马上想出最好的名字，有时在提炼的过程中好的名字才会出现。有时我会提炼一个函数，尝试使用它，然后发现不太合适，再把它内联回去，这完全没问题。只要在这个过程中学到了东西，我的时间就没有白费。

如果编程语言支持嵌套函数，就把新函数嵌套在源函数里，这能减少后面需要处理的超出作用域的变量个数。我可以稍后再使用搬移函数（198）把它从源函数中搬移出去。

- 将待提炼的代码从源函数复制到新建的目标函数中。
- 仔细检查提炼出的代码，看看其中是否引用了作用域限于源函数、在提炼出的新函数中访问不到的变量。若是，以参数的形式将它们传递给新函数。

如果提炼出的新函数嵌套在源函数内部，

就不存在变量作用域的问题了。

这些“作用域限于源函数”的变量通常是局部变量或者源函数的参数。最通用的做法是将它们都作为参数传递给新函数。只要没在提炼部分对这些变量赋值，处理起来就没什么难度。

如果某个变量是在提炼部分之外声明但只在提炼部分被使用，就把变量声明也搬移到提炼部分代码中去。

如果变量按值传递给提炼部分又在提炼部分被赋值，就必须多加小心。如果只有一个这样的变量，我会尝试将提炼出的新函数变成一个查询（query），用其返回值给该变量赋值。

但有时在提炼部分被赋值的局部变量太多，这时最好是先放弃提炼。这种情况下，我会考虑先使用别的重构手法，例如拆分变量（240）或者以查询取代临时变量（178），来简化变量的使用情况，然后再考虑提炼函数。

- 所有变量都处理完之后，编译。

如果编程语言支持编译期检查的话，在处理完所有变量之后做一次编译是很有用的，编译器经常会帮你找到没有被恰当处理的变量。

- 在源函数中，将被提炼代码段替换为对目标函数的调用。
- 测试。
- 查看其他代码是否有与被提炼的代码段相同或相似之处。如果有，考虑使用以函数调用取代内联代码（222）令其调用提炼出的新函数。

有些重构工具直接支持这一步。如果工具不支持，可以快速搜索一下，看看别处是否还有重复代码。

范例：无局部变量

在最简单的情况下，提炼函数易如反掌。请看下列函数：

```
function printOwing(invoice) {
  let outstanding = 0;
```

```
  console.log("***********************");
  console.log("**** Customer Owes ****");
  console.log("***********************");

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;

invoice.dueDate = new Date(today.getFullYear(), today.getMont

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);

console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

你可能会好奇`Clock.today`是干什么的。这是一个Clock Wrapper[mf-cw]，也就是封装系统时钟调用的对象。我尽量避免在代码中直接调用`Date.now()`这样的函数，因为这会导致测试行为不可预测，以及在诊断故障时难以复制出错时的情况。

我们可以轻松提炼出“打印横幅”的代码。我只需要剪切、粘贴再插入一个函数调用动作就行了：

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();
```

```
  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;

invoice.dueDate = new Date(today.getFullYear(), today.getMont

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);

console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
function printBanner() {
  console.log("***********************");
  console.log("**** Customer Owes ****");
  console.log("***********************");
}
```

同样，我还可以把“打印详细信息”部分也提炼出来：

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;

invoice.dueDate = new Date(today.getFullYear(), today.getMont

  printDetails();
```

```
  function printDetails() {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);

console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

看起来提炼函数是一个极其简单的重构。但很多时候，情况会变得比较复杂。

在上面的例子中，我把printDetails函数嵌套在printOwing函数内部，这样前者就能访问到printOwing内部定义的所有变量。如果我使用的编程语言不支持嵌套函数，就没法这样操作了，那么我就要面对“提炼出一个顶层函数”的问题。此时我必须细心处理“只存在于源函数作用域”的变量，包括源函数的参数以及源函数内部定义的临时变量。

范例：有局部变量

局部变量最简单的情况是：被提炼代码段只是读取这些变量的值，并不修改它们。这种情况下我可以简单地将它们当作参数传给目标函数。所以，如果我面对下列函数：

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;

invoice.dueDate = new Date(today.getFullYear(), today.getMont

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);

console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
```

就可以将“打印详细信息”这一部分提炼为带两个参数的函数：

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;

invoice.dueDate = new Date(today.getFullYear(), today.getMont

  printDetails(invoice, outstanding);
}
function printDetails(invoice, outstanding) {
```

```
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);

console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

如果局部变量是一个数据结构（例如数组、记录或者对象），而被提炼代码段又修改了这个结构中的数据，也可以如法炮制。所以，“设置到期日”的逻辑也可以用同样的方式提炼出来：

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function recordDueDate(invoice) {
  const today = Clock.today;

invoice.dueDate = new Date(today.getFullYear(), today.getMont
}
```

范例：对局部变量再赋值

如果被提炼代码段对局部变量赋值，问题就变得复杂了。这里我们只讨论临时变量的问题。如果你发现源函数的参数被赋值，应该马上使用拆分变量（240）将其变成临时变量。

被赋值的临时变量也分两种情况。较简单的情况是：这个变量只在被提炼代码段中使用。若果真如此，你可以将这个临时变量的声明移到被提炼代码段中，然后一起提炼出去。如果变量的初始化和使用离得有点儿远，可以用移动语句（223）把针对这个变量的操作放到一起。

比较糟糕的情况是：被提炼代码段之外的代码也使用了这个变量。此时我需要返回修改后的值。我会用下面这个已经很眼熟的函数来展示该怎么做：

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
```

前面的重构我都一步到位地展示了结果，因为它们都很简单。但这次我会一步一步展示“做法”里的每个步骤。

首先，把变量声明移动到使用处之前。

```
function printOwing(invoice) {
  printBanner();

  // calculate outstanding
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
```

然后把想要提炼的代码复制到目标函数中。

```
function printOwing(invoice) {
  printBanner();

  // calculate outstanding
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function calculateOutstanding(invoice) {
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }
```

```
  return outstanding;
}
```

由于outstanding变量的声明已经被搬移到提炼出的新函数中，就不需要再将其作为参数传入了。outstanding是提炼代码段中唯一被重新赋值的变量，所以我可以直接返回它。

我的JavaScript环境在编译期提供不了任何价值——简直还不如文本编辑器的语法分析有用，所以“做法”里的“编译”一步可以跳过了。下一件事是修改原来的代码，令其调用新函数。新函数返回了修改后的outstanding变量值，我需要将其存入原来的变量中。

```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding(invoice);
  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function calculateOutstanding(invoice) {
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }
  return outstanding;
}
```

在收工之前，我还要修改返回值的名字，使其符合我一贯的编码风格。

```
function printOwing(invoice) {
  printBanner();
  const outstanding = calculateOutstanding(invoice);
  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function calculateOutstanding(invoice) {
  let result = 0;
  for (const o of invoice.orders) {
    result += o.amount;
  }
  return result;
}
```

我还顺手把原来的outstanding变量声明成const的，令其在初始化之后不能再次被赋值。

这时候，你可能会问："如果需要返回的变量不止一个，又该怎么办呢？"

有几种选择。最好的选择通常是：挑选另一块代码来提炼。我比较喜欢让每个函数都只返回一个值，所以我会安排多个函数，用以返回多个值。如果真的有必要提炼一个函数并返回多个值，可以构造并返回一个记录对象—不过通常更好的办法还是回过头来重新处理局部变量，我常用的重构手法有以查询取代临时变量（178）和拆分变量（240）。

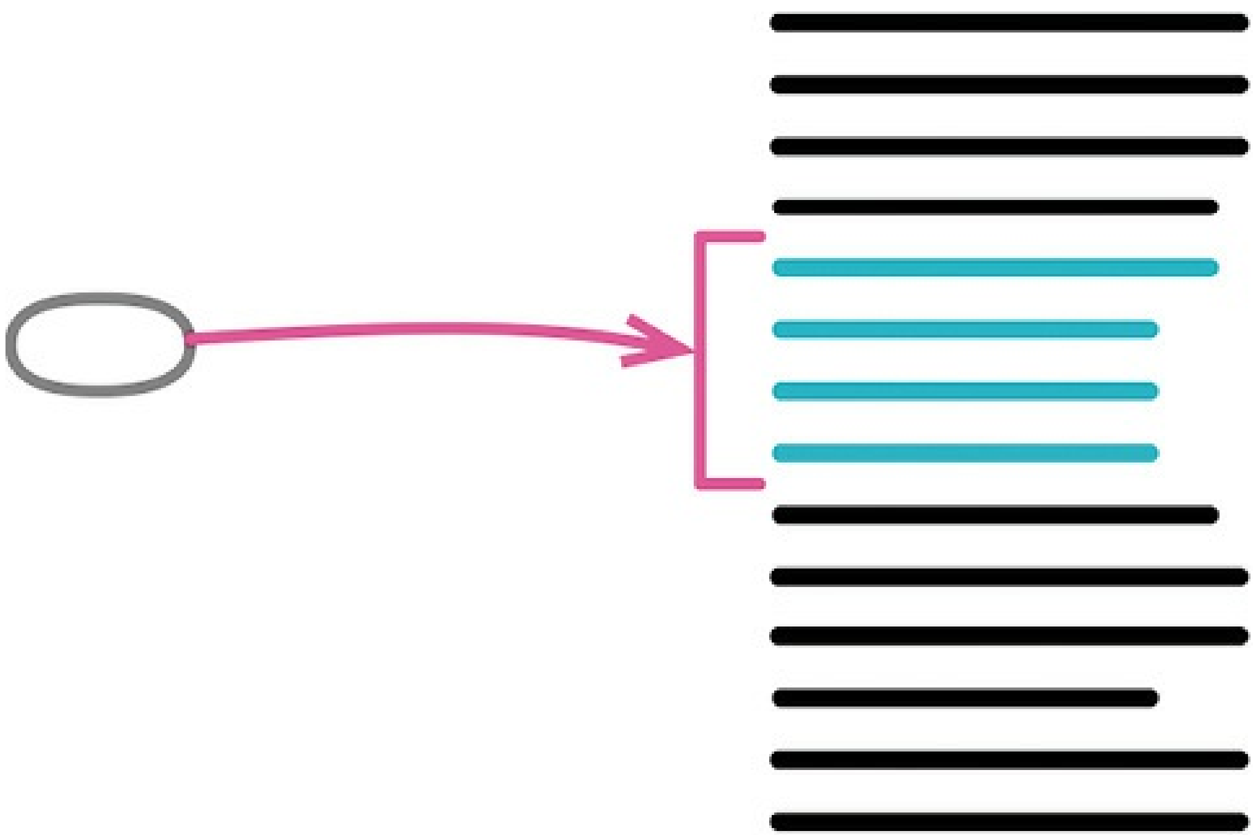
如果我想把提炼出的函数搬移到别的上下文（例如变成顶层函数），会引发一些有趣的问题。我偏好小步前进，所以我本能的做法是先提炼成嵌套函数，然后再将其移入新的上下文。但这种做法

的麻烦在于处理局部变量，而这个困难无法提前发现，直到我开始最后的搬移时才突然暴露。从这个角度考虑，即便可以先提炼成嵌套函数，或许也应该至少将目标函数放在源函数的同级，这样我就能立即看出提炼的范围是否合理。

6.2　内联函数（Inline Function）

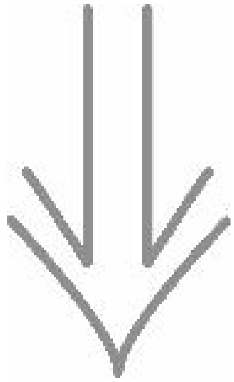
曾用名：内联函数（Inline Method）

反向重构：提炼函数（106）



```
function getRating(driver) {
  return moreThanFiveLateDeliveries(driver) ? 2 : 1;
}

function moreThanFiveLateDeliveries(driver) {
  return driver.numberOfLateDeliveries > 5;
}
```



```
function getRating(driver) {
  return (driver.numberOfLateDeliveries > 5) ? 2 : 1;
}
```

动机

本书经常以简短的函数表现动作意图，这样会使代码更清晰易读。但有时候你会遇到某些函数，其内部代码和函数名称同样清晰易读。也可能你重构了该函数的内部实现，使其内容和其名称变得同样清晰。若果真如此，你就应该去掉这个函数，直接使用其中的代码。间接性可能带来帮助，但非必要的间接性总是让人不舒服。

另一种需要使用内联函数的情况是：我手上有一群组织不甚合理的函数。可以将它们都内联到一个大型函数中，再以我喜欢的方式重新提炼出小函数。

如果代码中有太多间接层，使得系统中的所有函数都似乎只是对另一个函数的简单委托，造成我在这些委托动作之间晕头转向，那么我通常都会使用内联函数。当然，间接层有其价值，但不是所有间接层都有价值。通过内联手法，我可以找出那些有用的间接层，同时将无用的间接层去除。

做法

- 检查函数，确定它不具多态性。

如果该函数属于一个类，并且有子类继承了这个函数，那么就无法内联。

- 找出这个函数的所有调用点。
- 将这个函数的所有调用点都替换为函数本体。
- 每次替换之后，执行测试。

不必一次完成整个内联操作。如果某些调用点比较难以内联，可以等到时机成熟后再来

处理。

- 删除该函数的定义。

被我这样一写，内联函数似乎很简单。但情况往往并非如此。对于递归调用、多返回点、内联至另一个对象中而该对象并无访问函数等复杂情况，我可以写上好几页。我之所以不写这些特殊情况，原因很简单：如果你遇到了这样的复杂情况，就不应该使用这个重构手法。

范例

在最简单的情况下，这个重构简单得不值一提。一开始的代码是这样：

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}
function moreThanFiveLateDeliveries(aDriver) {
  return aDriver.numberOfLateDeliveries > 5;
}
```

我只要把被调用的函数的return语句复制出来，粘贴到调用处，取代原本的函数调用，就行

了。

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries &gt; 5 ? 2 : 1;
}
```

不过实际情况可能不会这么简单，需要我多做一点儿工作，帮助代码融入它的新家。例如，开始时的代码与前面稍有不同：

```
function rating(aDriver) {
 return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}

function moreThanFiveLateDeliveries(dvr) {
 return dvr.numberOfLateDeliveries > 5;
}
```

几乎是一样的代码，但`moreThanFiveLateDeliveries`函数声明的形式参数名与调用处使用的变量名不同，所以我在内联时需要对代码做些微调。

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

情况还可能更复杂。例如，请看下列代码：

```
function reportLines(aCustomer) {
  const lines = [];
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

我要把`gatherCustomerData`内联到`reportLines`中，这时简单的剪切和粘贴就不够了。这段代码还不算很麻烦，大多数时候我还是一步到位地完成了重构，只是需要做些调整。如果想更谨慎些，也可以每次搬移一行代码：可以首先对第一行代码使用搬移语句到调用者（217）——我还是用简单的“剪切-粘贴-调整”方式进行。

```
function reportLines(aCustomer) {
  const lines = [];
  lines.push(["name", aCustomer.name]);
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

然后继续处理后面的代码行，直到完成整个重构。

```
function reportLines(aCustomer) {
```

```
  const lines = [];
  lines.push(["name", aCustomer.name]);
  lines.push(["location", aCustomer.location]);
  return lines;
}
```

重点在于始终小步前进。大多数时候，由于我平时写的函数都很小，内联函数可以一步完成，顶多需要一点代码调整。但如果遇到了复杂的情况，我会每次内联一行代码。哪怕只是处理一行代码，也可能遇到麻烦，那么我就会使用更精细的重构手法搬移语句到调用者（217），将步子再拆细一点。有时我会自信满满地快速完成重构，然后测试却失败了，这时我会回退到上一个能通过测试的版本，带着一点儿懊恼，以更小的步伐再次重构。

6.3　提炼变量（Extract Variable）

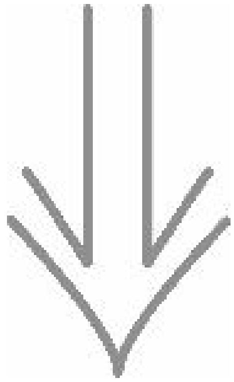
曾用名：引入解释性变量（Introduce Explaining Variable）

反向重构：内联变量（123）



```
return order.quantity * order.itemPrice -

Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
  Math.min(order.quantity * order.itemPrice * 0.1, 100);
```



```
const basePrice = order.quantity * order.itemPrice;
const quantityDiscount = Math.max(0, order.quantity - 500) *
const shipping = Math.min(basePrice * 0.1, 100);
return basePrice - quantityDiscount + shipping;
```

动机

表达式有可能非常复杂而难以阅读。这种情况下，局部变量可以帮助我们将表达式分解为比较容易管理的形式。在面对一块复杂逻辑时，局部变量使我能给其中的一部分命名，这样我就能更好地理解这部分逻辑是要干什么。

这样的变量在调试时也很方便，它们给调试器和打印语句提供了便利的抓手。

如果我考虑使用提炼变量，就意味着我要给代码中的一个表达式命名。一旦决定要这样做，我就

得考虑这个名字所处的上下文。如果这个名字只在当前的函数中有意义，那么提炼变量是个不错的选择；但如果这个变量名在更宽的上下文中也有意义，我就会考虑将其暴露出来，通常以函数的形式。如果在更宽的范围可以访问到这个名字，就意味着其他代码也可以用到这个表达式，而不用把它重写一遍，这样能减少重复，并且能更好地表达我的意图。

“将新的名字暴露得更宽”的坏处则是需要额外的工作量。如果工作量很大，我会暂时搁下这个想法，稍后再用以查询取代临时变量（178）来处理它。但如果处理其他很简单，我就会立即动手，这样马上就可以使用这个新名字。有一个好的例子：如果我处理的这段代码属于一个类，对这个新的变量使用提炼函数（106）会很容易。

做法

- 确认要提炼的表达式没有副作用。
- 声明一个不可修改的变量，把你想要提炼的表达式复制一份，以该表达式的结果值给这个变量赋值。
- 用这个新变量取代原来的表达式。

- 测试。

如果该表达式出现了多次，请用这个新变量逐一替换，每次替换之后都要执行测试。

范例

我们从一个简单计算开始：

```
function price(order) {
  //price is base price - quantity discount + shipping
  return order.quantity * order.itemPrice -

Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

这段代码还算简单，不过我可以让它变得更容易理解。首先，我发现，底价（base price）等于数量（quantity）乘以单价（item price）。

```
function price(order) {
  //price is base price - quantity discount + shipping
  return order.quantity * order.itemPrice -

Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

我把这一新学到的知识放进代码里，创建一个变量，并给它起个合适的名字：

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;
  return order.quantity * order.itemPrice -

Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

当然，仅仅声明并初始化一个变量没有任何作用，我还得使用它才行。所以，我用这个变量取代了原来的表达式：

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -

Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

稍后的代码还用到了同样的表达式，也可以用新建的变量取代之。

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -
```

```
Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(basePrice * 0.1, 100);
}
```

下一行是计算批发折扣（quantity discount）的逻辑，我也将它提炼出来：

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;

const quantityDiscount = Math.max(0, order.quantity - 500) *
  return basePrice -
    quantityDiscount +
    Math.min(basePrice * 0.1, 100);
}
```

最后，我再把运费（shipping）计算提炼出来。同时我还可以删掉代码中的注释，因为现在代码已经可以完美表达自己的意义了：

```
function price(order) {
  const basePrice = order.quantity * order.itemPrice;

const quantityDiscount = Math.max(0, order.quantity - 500) *
  const shipping = Math.min(basePrice * 0.1, 100);
  return basePrice - quantityDiscount + shipping;
}
```

范例：在一个类中

下面是同样的代码，但这次它位于一个类中：

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }


  get quantity() {return this._data.quantity;}
  get itemPrice() {return this._data.itemPrice;}

  get price() {
    return this.quantity * this.itemPrice -

Math.max(0, this.quantity - 500) * this.itemPrice * 0.05 +
      Math.min(this.quantity * this.itemPrice * 0.1, 100);
  }
}
```

我要提炼的还是同样的变量，但我意识到：这些变量名所代表的概念，适用于整个Order类，而不仅仅是“计算价格”的上下文。既然如此，我更愿意将它们提炼成方法，而不是变量。

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }
  get quantity() {return this._data.quantity;}
  get itemPrice() {return this._data.itemPrice;}

  get price() {

return this.basePrice - this.quantityDiscount + this.shipping
  }

get basePrice()        {return this.quantity * this.itemPrice

get quantityDiscount() {return Math.max(0, this.quantity - 50
```

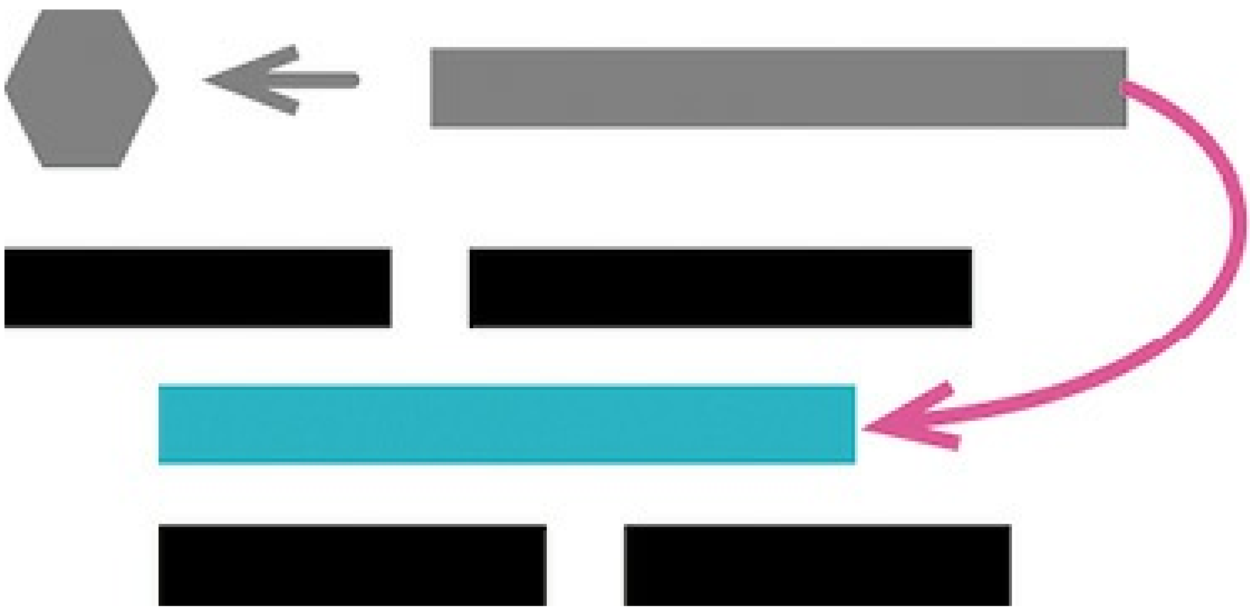
```
get shipping()        {return Math.min(this.basePrice * 0.1,
}
```

这是对象带来的一大好处：它们提供了合适的上下文，方便分享相关的逻辑和数据。在如此简单的情况下，这方面的好处还不太明显；但在一个更大的类当中，如果能找出可以共用的行为，赋予它独立的概念抽象，给它起一个好名字，对于使用对象的人会很有帮助。

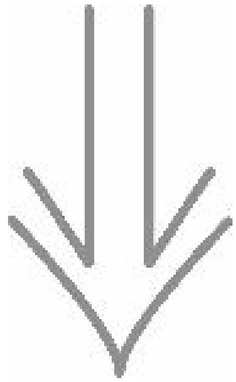
6.4 内联变量（Inline Variable）

曾用名：内联临时变量（Inline Temp）

反向重构：提炼变量（119）



```
let basePrice = anOrder.basePrice;
return (basePrice > 1000);
```



```
return anOrder.basePrice > 1000;
```

动机

在一个函数内部，变量能给表达式提供有意义的名字，因此通常变量是好东西。但有时候，这个名字并不比表达式本身更具表现力。还有些时候，变量可能会妨碍重构附近的代码。若果真如此，就应该通过内联的手法消除变量。

做法

- 检查确认变量赋值语句的右侧表达式没有副作用。

- 如果变量没有被声明为不可修改，先将其变为不可修改，并执行测试。

这是为了确保该变量只被赋值一次。

- 找到第一处使用该变量的地方，将其替换为直接使用赋值语句的右侧表达式。
- 测试。
- 重复前面两步，逐一替换其他所有使用该变量的地方。
- 删除该变量的声明点和赋值语句。
- 测试。

6.5　改变函数声明（Change Function Declaration）

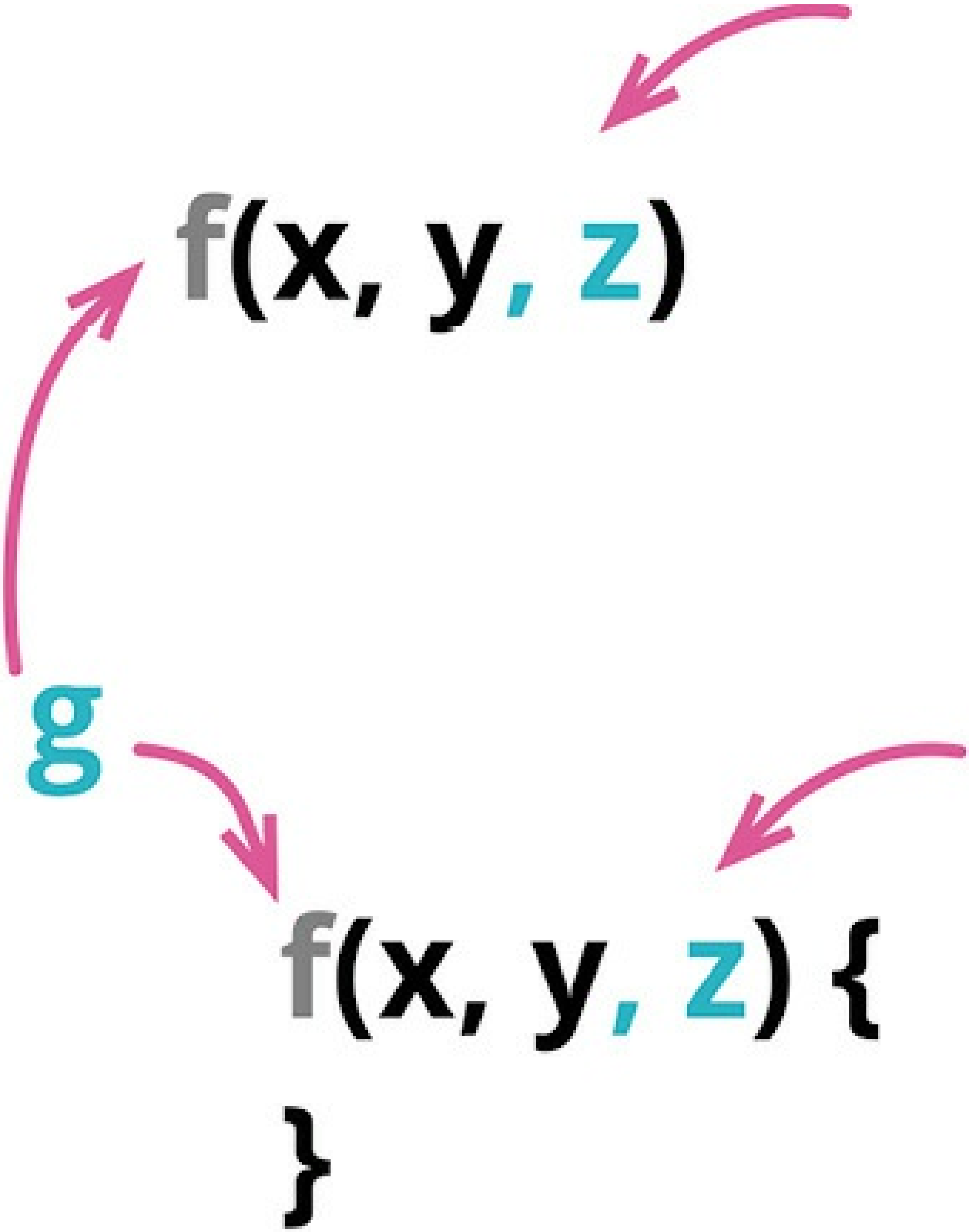
别名：函数改名（Rename Function）

曾用名：函数改名（Rename Method）

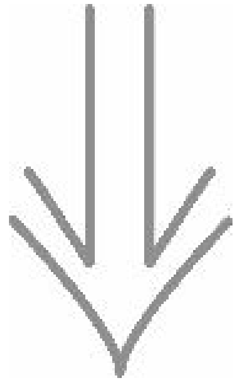
曾用名：添加参数（Add Parameter）

曾用名：移除参数（Remove Parameter）

别名：修改签名（Change Signature）


f(x, y, z)
g
f(x, y, z) {
}

```
function circum(radius) {...}
```



```
function circumference(radius) {...}
```

动机

函数是我们将程序拆分成小块的主要方式。函数声明则展现了如何将这些小块组合在一起工作——可以说，它们就是软件系统的关节。和任何构造体一样，系统的好坏很大程度上取决于关节。好的关节使得给系统添加新部件很容易；而糟糕的关节则不断招致麻烦，让我们难以看清软件的行为，当需求变化时难以找到合适的地方进行修改。还好，软件是软的，我可以改变这些关节，只是要小心修改。

对于这些关节而言，最重要的元素当属函数的名字。一个好名字能让我一眼看出函数的用途，而不必查看其实现代码。但起一个好名字并不容易，我很少能第一次就把名字起对。“就算这个名字有点迷惑人，还是放着别管吧——说到底，不过就是一个名字而已。”邪恶的混乱魔王就是这样引诱我的。为了拯救程序的灵魂，绝不能上了他的当。如果我看到一个函数的名字不对，一旦发现了更好的名字，就得尽快给函数改名。这样，下一次再看到这段代码时，我就不用再费力搞懂其中到底在干什么。（有一个改进函数名字的好办法：先写一句注释描述这个函数的用途，再把这句注释变成函数的名字。）

对于函数的参数，道理也是一样。函数的参数列表阐述了函数如何与外部世界共处。函数的参数设置了一个上下文，只有在这个上下文中，我才能使用这个函数。假如有一个函数的用途是把某人的电话号码转换成特定的格式，并且该函数的参数是一个人（person），那么我就没法用这个函数来处理公司（company）的电话号码。如果我把函数接受的参数由“人”改成“电话号码”，这段处理电话号码格式的代码就能被更广泛地使用。

修改参数列表不仅能增加函数的应用范围，还能改变连接一个模块所需的条件，从而去除不必要

的耦合。在前面这个例子中，修改参数列表之后，“处理电话号码格式”的逻辑所在的模块就无须了解“人”这个概念。减少模块彼此之间的信息依赖，当我要做出修改时就能减轻我大脑的负担——毕竟我的脑容量已经不如从前那么大了（跟我脑袋的大小没关系）。

如何选择正确的参数，没有简单的规则可循。我可能有一个简单的函数，用于判断支付是否逾期——如果超期30天未付款，那么这笔支付就逾期了。这个函数的参数应该是“支付”（payment）对象，还是支付的到期日呢？如果使用支付对象，会使这个函数与支付对象的接口耦合，但好处是可以很容易地访问后者的其他属性，当“逾期”的逻辑发生变化时就不用修改所有调用该函数的代码——换句话说，提高了该函数的封装度。

对这道难题，唯一正确的答案是“没有正确答案”，而且答案还会随着时间变化。所以我发现掌握改变函数声明重构手法至关重要，这样当我想好代码中应该有哪些关节时，才能使代码随着我的理解而演进。

在本书中引用重构手法时，我通常只使用它的主名称。但“改名”（rename）是改变函数声明的重要应用场景，所以，如果只是用于改名，我会将这个重构称作函数改名（Rename Function），这样能

更清晰地表达我的用意。从做法的角度，不管是给函数改名还是修改参数列表，做法都是一样的。

做法

对于本书中的大部分重构，我只展示了一套做法。这并非因为只有这一套做法，而是因为大部分情况下，一套标准的做法都管用。不过，改变函数声明是一个例外。它有一套简单的做法，这套做法常常够用；但在很多时候，有必要以更渐进的方式逐步迁移到达最终结果。所以，在进行此重构时，我会查看变更的范围，自问是否能一步到位地修改函数声明及其所有调用者。如果可以，我就采用简单的做法。迁移式的做法让我可以逐步修改调用方代码，如果函数被很多地方调用，或者修改不容易，或者要修改的是一个多态函数，或者对函数声明的修改比较复杂，能渐进式地逐步修改就很重要。

简单的做法

- 如果想要移除一个参数，需要先确定函数体内没有使用该参数。

- 修改函数声明，使其成为你期望的状态。
- 找出所有使用旧的函数声明的地方，将它们改为使用新的函数声明。
- 测试。

最好能把大的修改拆成小的步骤，所以如果你既想修改函数名，又想添加参数，最好分成两步来做。（并且，不论何时，如果遇到了麻烦，请撤销修改，并改用迁移式做法。）

迁移式做法

- 如果有必要的话，先对函数体内部加以重构，使后面的提炼步骤易于开展。
- 使用提炼函数（106）将函数体提炼成一个新函数。

  如果你打算沿用旧函数的名字，可以先给新函数起一个易于搜索的临时名字。

- 如果提炼出的函数需要新增参数，用前面的简单做法添加即可。
- 测试。

- 对旧函数使用内联函数（115）。
- 如果新函数使用了临时的名字，再次使用改变函数声明（124）将其改回原来的名字。
- 测试。

如果要重构的函数属于一个具有多态性的类，那么对于该函数的每个实现版本，你都需要通过“提炼出一个新函数”的方式添加一层间接，并把旧函数的调用转发给新函数。如果该函数的多态性是在一个类继承体系中体现，那么只需要在超类上转发即可；如果各个实现类之间并没有一个共同的超类，那么就需要在每个实现类上做转发。

如果要重构一个已对外发布的API，在提炼出新函数之后，你可以暂停重构，将原来的函数声明为“不推荐使用”（`deprecated`），然后给客户端一点时间转为使用新函数。等你有信心所有客户端都已经从旧函数迁移到新函数，再移除旧函数的声明。

范例：函数改名（简单做法）

下列函数的名字太过简略了：

```
function circum(radius) {
```

```
    return 2 * Math.PI * radius;
  }
```

我想把它改得更有意义一点儿。首先修改函数的声明：

```
  function circumference(radius) {
    return 2 * Math.PI * radius;
  }
```

然后找出所有调用circum函数的地方，将其改为circumference。

在不同的编程语言环境中，“找到所有调用旧函数的地方”这件事的难度也各异。静态类型加上趁手的IDE能提供最好的体验，通常可以全自动地完成函数改名，出错的概率极低。如果没有静态类型，就需要多花些工夫：即便再好的搜索工具，也可能会找出很多同名但并非同一函数的地方。

增减参数的做法也相同：找出所有调用者，修改函数声明，然后修改调用者。最好是能分步骤修改：如果既想给函数改名，又想添加参数，我会先完成改名，测试，然后添加参数，然后再次测试。

这个重构的简单做法缺点在于，我必须一次性修改所有调用者和函数声明（或者说，所有的函数

声明，如果有多态的话）。如果只有不多的几处调用者，或者如果有可靠的自动化重构工具，这样做是没问题的。但如果调用者很多，事情就会变得很棘手。另外，如果函数的名字并不唯一，也可能造成问题。例如，我想给代表“人”的Person类的changeAddress函数改名，但同时在代表“保险合同”的InsuranceAgreement类中也有一个同名的函数，而我并不想修改后者的名字。修改越是复杂，我就越不希望一步到位地完成。如果有这些问题出现，我就会改为使用迁移式做法。同样，如果使用简单做法时出了什么错，我也会把代码回滚到上一个已知正确的状态，并改用迁移式做法再来一遍。

范例：函数改名（迁移式做法）

还是这个名字太过简略的函数：

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

按照迁移式做法，我首先要对整个函数体使用提炼函数（106）：

```
function circum(radius) {
  return circumference(radius);
}
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

此时我要执行测试，然后对旧函数使用内联函数（115）：找出所有调用旧函数的地方，将其改为调用新函数。每次修改之后都可以执行测试，这样我就可以小步前进，每次修改一处调用者。所有调用者都修改完之后，我就可以删除旧函数。

大多数重构手法只用于修改我有权修改的代码，但这个重构手法同样适用于已发布API——使用这些API的代码我无权修改。以上面的代码为例，创建出`circumference`函数之后，我就可以暂停重构，并（如果可以的话）将`circum`函数标记为`deprecated`。然后我就耐心等待客户端改用`circumference`函数，等他们都改完了，我再删除`circum`函数。即便永远也抵达不了“删除`circum`函数”这个快乐的终点，至少新代码有了一个更好的名字。

范例：添加参数

想象一个管理图书馆的软件，其中有代表“图书”的Book类，它可以接受顾客（customer）的预订（reservation）：

class Book...

```
addReservation(customer) {
  this._reservations.push(customer);
}
```

现在我需要支持“高优先级预订”，因此我要给addReservation额外添加一个参数，用于标记这次预订应该进入普通队列还是优先队列。如果能很容易地找到并修改所有调用方，我可以直接修改；但如果不行，我仍然可以采用迁移式做法，下面是详细的过程。

首先，我用提炼函数（106）把addReservation的函数体提炼出来，放进一个新函数。这个新函数最终会叫addReservation，但新旧两个函数不能同时占用这个名字，所以我会先给新函数起一个容易搜索的临时名字。

class Book...

```
addReservation(customer) {
  this.zz_addReservation(customer);
}
zz_addReservation(customer) {
  this._reservations.push(customer);
}
```

然后我会在新函数的声明中增加参数，同时修改旧函数中调用新函数的地方（也就是采用简单做法完成这一步）。

class Book...

```
addReservation(customer) {
  this.zz_addReservation(customer, false);
}

zz_addReservation(customer, isPriority) {
  this._reservations.push(customer);
}
```

在修改调用方之前，我喜欢利用JavaScript的语言特性先应用引入断言（302），确保调用方一定会用到这个新参数。

class Book...

```
zz_addReservation(customer, isPriority) {
  assert(isPriority === true || isPriority === false);
```

```
    this._reservations.push(customer);
  }
```

现在，如果我在修改调用方时出了错，没有提供新参数，这个断言会帮我抓到错误——以我过去的经验来看，比我更容易出错的程序员怕是不多。

现在，我可以对源函数使用内联函数（115），使其调用者转而使用新函数。这样我可以每次只修改一个调用者。

现在我就可以把新函数改回原来的名字了。一般而言，此时用简单做法就够了；但如果有必要，也可以再用一遍迁移式做法。

范例：把参数改为属性

此前的范例都很简单：改个名，增加一个参数。有了迁移式做法以后，这个重构手法可以相当利落地处理更复杂的情况。下面就是一个更复杂的例子。

假设我有一个函数，用于判断顾客（customer）是不是来自新英格兰（New England）地区：

```
function inNewEngland(aCustomer) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(aCusto
}
```

下面是一个调用该函数的地方：

调用方...

```
const newEnglanders = someCustomers.filter(c => inNewEngland(
```

`inNewEngland`函数只用到了顾客所在的州（state）这项信息，基于这个信息来判断顾客是否来自新英格兰地区。我希望重构这个函数，使其接受州代码（state code）作为参数，这样就能去掉对“顾客”概念的依赖，使这个函数能在更多的上下文中使用。

在使用改变函数声明时，我通常会先运用提炼函数（106），但在这里我会先对函数体做一点重构，使后面的重构步骤更简单。我先用提炼变量（119）提炼出我想要的新参数：

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateC
}
```

然后再用提炼函数（106）创建新函数：

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return xxNEWinNewEngland(stateCode);
}

function xxNEWinNewEngland(stateCode) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateC
}
```

我会给新函数起一个好记又独特的临时名字，这样回头要改回原来的名字时也会简单一些。（你也看到，对于怎么起这些临时名字，我并没有统一的标准。）

我会在源函数中使用内联变量（123），把刚才提炼出来的参数内联回去：

```
function inNewEngland(aCustomer) {
  return xxNEWinNewEngland(aCustomer.address.state);
}
```

然后我会用内联函数（115）把旧函数内联到调用处，其效果就是把旧函数的调用处改为调用新函数。我可以每次修改一个调用处。

调用方...

```
const newEnglanders = someCustomers.filter(c => xxNEWinNewEng
```

旧函数被内联到各调用处之后，我就再次使用改变函数声明，把新函数改回旧名字：

调用方...

```
const newEnglanders = someCustomers.filter(c => inNewEngland(
```

顶层作用域...

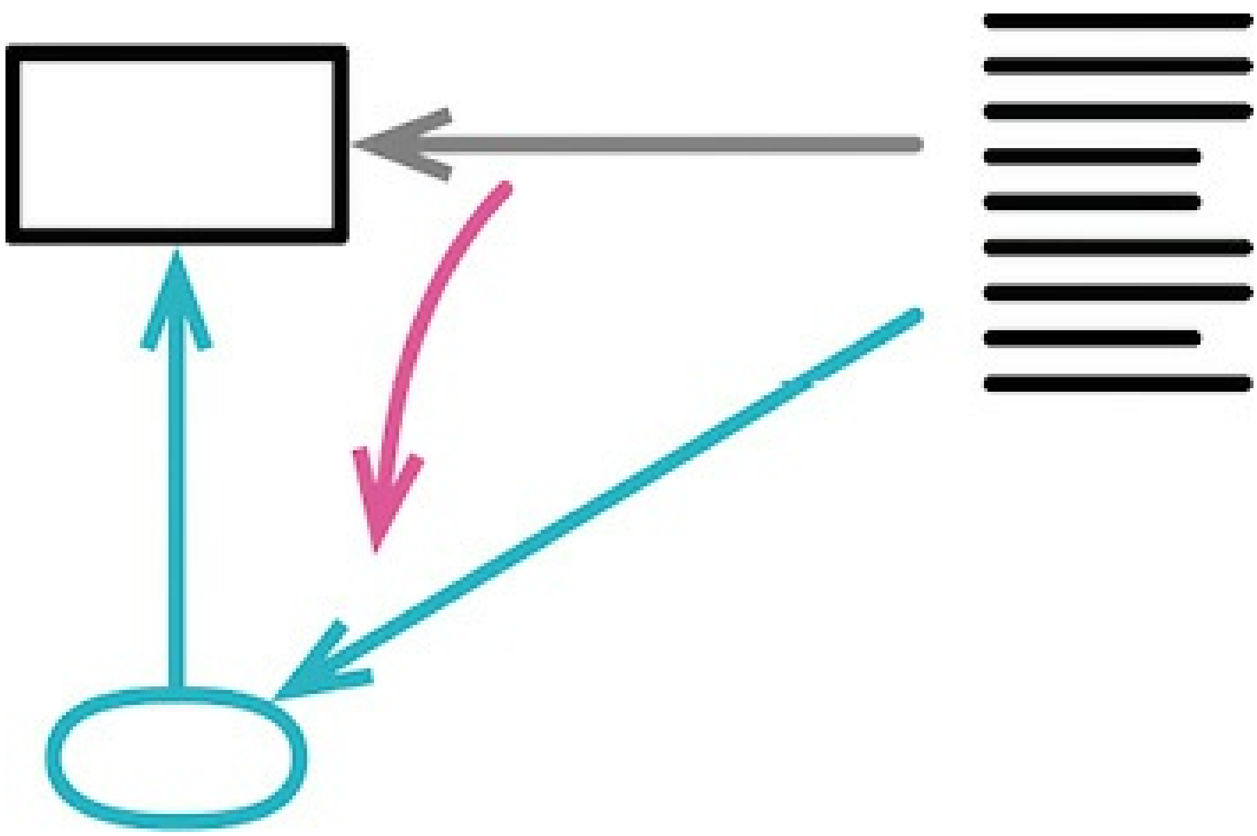
```
function inNewEngland(stateCode) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateC
}
```

自动化重构工具减少了迁移式做法的用武之地，同时也使迁移式做法更加高效。自动化重构工具可以安全地处理相当复杂的改名、参数变更等情况，所以迁移式做法的用武之地就变少了，因为自动化重构工具经常能提供足够的支持。如果遇到类似这里的例子，尽管工具无法自动完成整个重构，还是可以更快、更安全地完成关键的提炼和内联步骤，从而简化整个重构过程。

6.6 封装变量（Encapsulate Variable）

曾用名：自封装字段（Self-Encapsulate Field）

曾用名：封装字段（Encapsulate Field）



```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowle
export function defaultOwner()       {return defaultOwnerData
export function setDefaultOwner(arg) {defaultOwnerData = arg;
```

动机

重构的作用就是调整程序中的元素。函数相对容易调整一些，因为函数只有一种用法，就是调用。在改名或搬移函数的过程中，总是可以比较容易地保留旧函数作为转发函数（即旧代码调用旧函数，旧函数再调用新函数）。这样的转发函数通常不会存在太久，但的确能够简化重构过程。

数据就要麻烦得多，因为没办法设计这样的转发机制。如果我把数据搬走，就必须同时修改所有引用该数据的代码，否则程序就不能运行。如果数据的可访问范围很小，比如一个小函数内部的临时变量，那还不成问题。但如果可访问范围变大，重

构的难度就会随之增大，这也是说全局数据是大麻烦的原因。

所以，如果想要搬移一处被广泛使用的数据，最好的办法往往是先以函数形式封装所有对该数据的访问。这样，我就能把“重新组织数据”的困难任务转化为“重新组织函数”这个相对简单的任务。

封装数据的价值还不止于此。封装能提供一个清晰的观测点，可以由此监控数据的变化和使用情况；我还可以轻松地添加数据被修改时的验证或后续逻辑。我的习惯是：对于所有可变的数据，只要它的作用域超出单个函数，我就会将其封装起来，只允许通过函数访问。数据的作用域越大，封装就越重要。处理遗留代码时，一旦需要修改或增加使用可变数据的代码，我就会借机把这份数据封装起来，从而避免继续加重耦合一份已经广泛使用的数据。

面向对象方法如此强调对象的数据应该保持私有（private），背后也是同样的原理。每当看见一个公开（public）的字段时，我就会考虑使用封装变量（在这种情况下，这个重构手法常被称为封装字段）来缩小其可见范围。一些更激进的观点认为，即便在类内部，也应该通过访问函数来使用字段——这种做法也称为“自封装”。大体而言，我认为自封装有点儿过度了——如果一个类大到需要将

字段自封装起来的程度，那么首先应该考虑把这个类拆小。不过，在分拆类之前，自封装字段倒是一个有用的步骤。

封装数据很重要，不过，不可变数据更重要。如果数据不能修改，就根本不需要数据更新前的验证或者其他逻辑钩子。我可以放心地复制数据，而不用搬移原来的数据——这样就不用修改使用旧数据的代码，也不用担心有些代码获得过时失效的数据。不可变性是强大的代码防腐剂。

做法

- 创建封装函数，在其中访问和更新变量值。
- 执行静态检查。
- 逐一修改使用该变量的代码，将其改为调用合适的封装函数。每次替换之后，执行测试。
- 限制变量的可见性。

有时没办法阻止直接访问变量。若果真如此，可以试试将变量改名，再执行测试，找出仍在直接使用该变量的代码。

- 测试。
- 如果变量的值是一个记录，考虑使用封装记录（162）。

范例

下面这个全局变量中保存了一些有用的数据：

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```

使用它的代码平淡无奇：

```
spaceship.owner = defaultOwner;
```

更新这段数据的代码是这样：

```
defaultOwner = {firstName: "Rebecca", lastName: "Parsons"};
```

首先我要定义读取和写入这段数据的函数，给它做个基础的封装。

```
function getDefaultOwner() {return defaultOwner;}
function setDefaultOwner(arg) {defaultOwner = arg;}
```

然后就开始处理使用defaultOwner的代码。每看见一处引用该数据的代码，就将其改为调用取值函数。

```
spaceship.owner = getDefaultOwner();
```

每看见一处给变量赋值的代码，就将其改为调用设值函数。

```
setDefaultOwner({firstName: "Rebecca", lastName: "Parsons"});
```

每次替换之后，执行测试。

处理完所有使用该变量的代码之后，我就可以限制它的可见性。这一步的用意有两个，一来是检查是否遗漏了变量的引用，二来可以保证以后的代码也不会直接访问该变量。在JavaScript中，我可以把变量和访问函数搬移到单独一个文件中，并且只导出访问函数，这样就限制了变量的可见性。

defaultOwner.js...

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
export function getDefaultOwner()    {return defaultOwner;}
export function setDefaultOwner(arg) {defaultOwner = arg;}
```

如果条件不允许限制对变量的访问，可以将变量改名，然后再次执行测试，检查是否仍有代码在直接使用该变量。这阻止不了未来的代码直接访问变量，不过可以给变量起个有意义又难看的名字（例如__privateOnly_defaultOwner），提醒后来的客户端。

我不喜欢给取值函数加上get前缀，所以我对这个函数改名。

defaultOwner.js...

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowle
export function getdefaultOwner()     {return defaultOwnerDa
export function setDefaultOwner(arg) {defaultOwnerData = arg;
```

JavaScript有一种惯例：给取值函数和设值函数起同样的名字，根据有没有传入参数来区分。我把这种做法称为“重载取值/设值函数”（Overloaded Getter Setter）[mf-orgs]，并且我强烈反对这种做法。所以，虽然我不喜欢get前缀，但我会保留set前缀。

封装值

前面介绍的基本重构手法对数据结构的引用做了封装，使我能控制对该数据结构的访问和重新赋值，但并不能控制对结构内部数据项的修改：

```
const owner1 = defaultOwner();
assert.equal("Fowler", owner1.lastName, "when set");
const owner2 = defaultOwner();
owner2.lastName = "Parsons";
assert.equal("Parsons", owner1.lastName, "after change owner2
```

前面的基本重构手法只封装了对最外层数据的引用。很多时候这已经足够了。但也有很多时候，我需要把封装做得更深入，不仅控制对变量引用的修改，还要控制对变量内容的修改。

这有两个办法可以做到。最简单的办法是禁止对数据结构内部的数值做任何修改。我最喜欢的一种做法是修改取值函数，使其返回该数据的一份副本。

defaultOwner.js...

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowle
export function defaultOwner()      {return Object.assign({}
```

```
export function setDefaultOwner(arg) {defaultOwnerData = arg;
```

对于列表数据，我尤其常用这一招。如果我在取值函数中返回数据的一份副本，客户端可以随便修改它，但不会影响到共享的这份数据。但在使用副本的做法时，我必须格外小心：有些代码可能希望能修改共享的数据。若果真如此，我就只能依赖测试来发现问题了。另一种做法是阻止对数据的修改，比如通过封装记录（162）就能很好地实现这一效果。

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowle
export function defaultOwner()       {return new Person(defau
export function setDefaultOwner(arg) {defaultOwnerData = arg;

class Person {
  constructor(data) {
    this._lastName = data.lastName;
    this._firstName = data.firstName
  }
  get lastName() {return this._lastName;}
  get firstName() {return this._firstName;}
  // and so on for other properties
```

现在，如果客户端调用`defaultOwner`函数获得“默认拥有人”数据、再尝试对其属性（即`lastName`和`firstName`）重新赋值，赋值不会产生任何效果。对于侦测或阻止修改数据结构内部的数据项，各种编程语言有不同的方式，所以我会根据当下使用的语言来选择具体的办法。

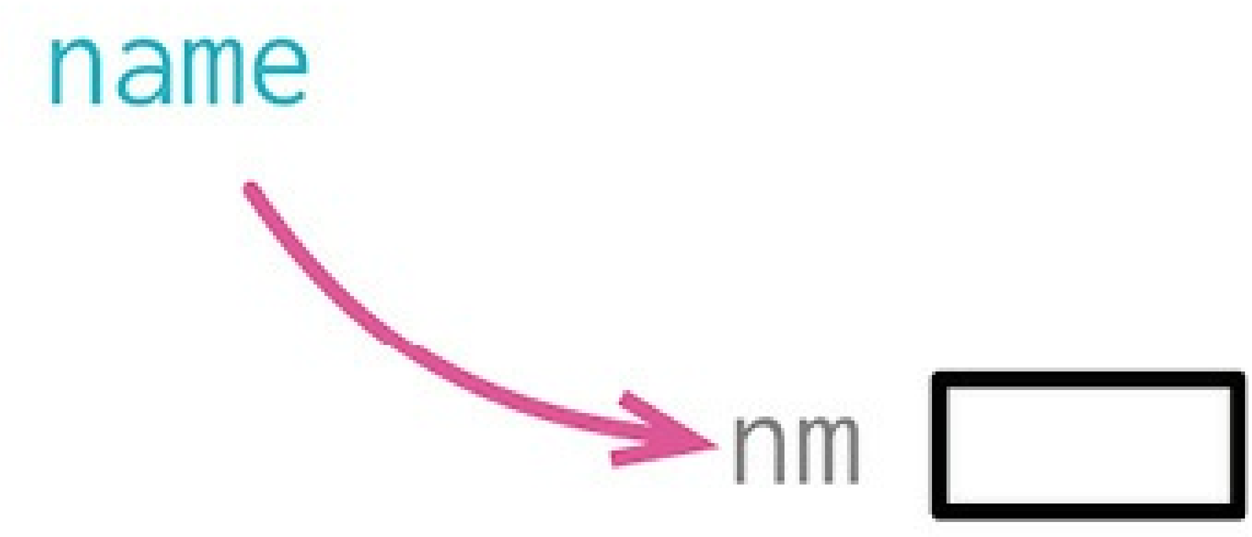
“侦测和阻止修改数据结构内部的数据项”通常只是个临时处置。随后我可以去除这些修改逻辑，或者提供适当的修改函数。这些都处理完之后，我就可以修改取值函数，使其返回一份数据副本。

到目前为止，我都在讨论“在取数据时返回一份副本”，其实设值函数也可以返回一份副本。这取决于数据从哪儿来，以及我是否需要保留对源数据的连接，以便知悉源数据的变化。如果不需要这样一条连接，那么设值函数返回一份副本就有好处：可以防止因为源数据发生变化而造成的意外事故。很多时候可能没必要复制一份数据，不过多一次复制对性能的影响通常也都可以忽略不计。但是，如果不做复制，风险则是未来可能会陷入漫长而困难的调试排错过程。

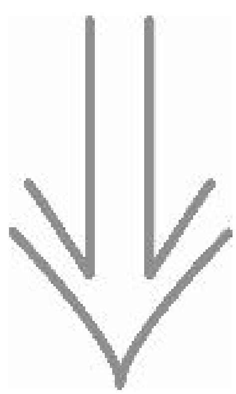
请记住，前面提到的数据复制、类封装等措施，都只在数据记录结构中深入了一层。如果想走得更深入，就需要更多层级的复制或是封装。

如你所见，数据封装很有价值，但往往并不简单。到底应该封装什么，以及如何封装，取决于数据被使用的方式，以及我想要修改数据的方式。不过，一言以蔽之，数据被使用得越广，就越是值得花精力给它一个体面的封装。

6.7 变量改名（Rename Variable）




```
let a = height * width;
```



```
let area = height * width;
```

动机

好的命名是整洁编程的核心。变量可以很好地解释一段程序在干什么——如果变量名起得好的话。但我经常会把名字起错——有时是因为想得不够仔细，有时是因为我对问题的理解加深了，还有时是因为程序的用途随着用户的需求改变了。

使用范围越广，名字的好坏就越重要。只在一行的lambda表达式中使用的变量，跟踪起来很容易——像这样的变量，我经常只用一个字母命名，因为变量的用途在这个上下文中很清晰。同理，短函数的参数名也常常很简单。不过在JavaScript这样的动态类型语言中，我喜欢把类型信息也放进名字里（于是变量名可能叫`aCustomer`）。

对于作用域超出一次函数调用的字段，则需要更用心命名。这是我最花心思的地方。

机制

- 如果变量被广泛使用，考虑运用封装变量（132）将其封装起来。
- 找出所有使用该变量的代码，逐一修改。

如果在另一个代码库中使用了该变量，这就是一个“已发布变量”（published variable），此时不能进行这个重构。

如果变量值从不修改，可以将其复制到一个新名字之下，然后逐一修改使用代码，每次修改后执行测试。

- 测试。

范例

如果要改名的变量只作用于一个函数（临时变量或者参数），对其改名是最简单的。这种情况太简单，根本不需要范例：找到变量的所有引用，修改过来就行。完成修改之后，我会执行测试，确保没有破坏什么东西。

如果变量的作用域不止于单个函数，问题就会出现。代码库的各处可能有很多地方使用它：

```
let tpHd = "untitled";
```

有些地方是在读取变量值：

```
result += `<h1>${tpHd}</h1>`;
```

另一些地方则更新它的值：

```
tpHd = obj['articleTitle'];
```

对于这种情况，我通常的反应是运用封装变量（132）：

```
result += `<h1>${title()}</h1>`;

setTitle(obj['articleTitle']);

  function title()       {return tpHd;}
  function setTitle(arg) {tpHd = arg;}
```

现在就可以给变量改名：

```
let _title = "untitled";
```

```
function title()       {return _title;}
function setTitle(arg) {_title = arg;}
```

我可以继续重构下去，将包装函数内联回去，这样所有的调用者就变回直接使用变量的状态。不过我很少这样做。如果这个变量被广泛使用，以至于我感到需要先做封装才敢改名，那就有必要保持这个状态，将变量封装在函数后面。

> 如果我确实想内联，在重构过程中，我就会将取值函数命名为getTitle，并且其中的变量名也不会以下划线开头。

给常量改名

如果我想改名的是一个常量（或者在客户端看来就像是常量的元素），我可以复制这个常量，这样既不需要封装，又可以逐步完成改名。假如原来的变量声明是这样：

```
const cpyNm = "Acme Gooseberries";
```

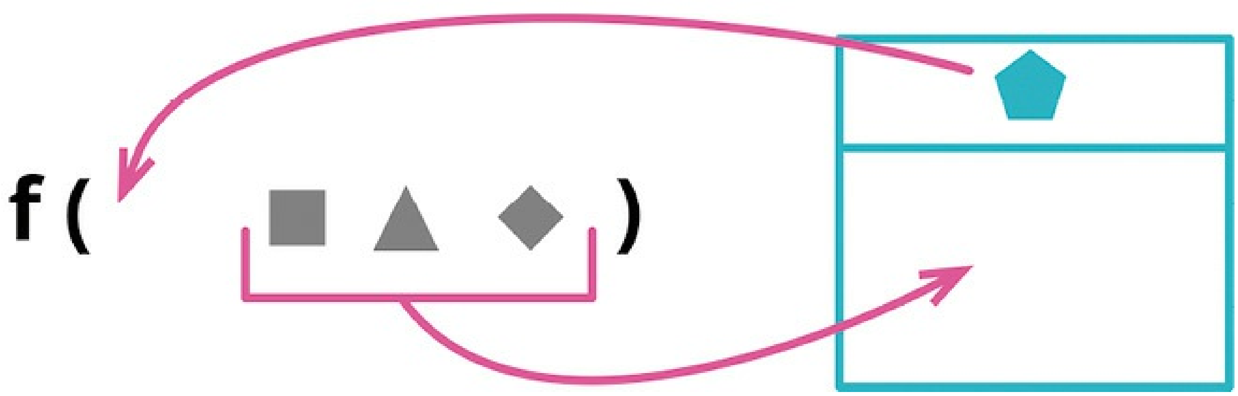
改名的第一步是复制这个常量：

```
const companyName = "Acme Gooseberries";
const cpyNm = companyName;
```

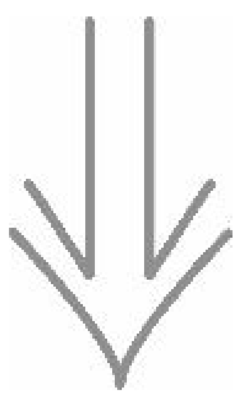
有了这个副本，我就可以逐一修改引用旧常量的代码，使其引用新的常量。全部修改完成后，我会删掉旧的常量。我喜欢先声明新的常量名，然后把新常量复制给旧的名字。这样最后删除旧名字时会稍微容易一点，如果测试失败，再把旧常量放回来也稍微容易一点。

这个做法不仅适用于常量，也同样适用于客户端只能读取的变量（例如JavaScript模块中导出的变量）。

6.8　引入参数对象（Introduce Parameter Object）



```
function amountInvoiced(startDate, endDate) {...}
function amountReceived(startDate, endDate) {...}
function amountOverdue(startDate, endDate) {...}
```



```
function amountInvoiced(aDateRange) {...}
function amountReceived(aDateRange) {...}
function amountOverdue(aDateRange) {...}
```

动机

我常会看见，一组数据项总是结伴同行，出没于一个又一个函数。这样一组数据就是所谓的数据泥团，我喜欢代之以一个数据结构。

将数据组织成结构是一件有价值的事，因为这让数据项之间的关系变得明晰。使用新的数据结构，参数的参数列表也能缩短。并且经过重构之后，所有使用该数据结构的函数都会通过同样的名字来访问其中的元素，从而提升代码的一致性。

但这项重构真正的意义在于，它会催生代码中更深层次的改变。一旦识别出新的数据结构，我就可以重组程序的行为来使用这些结构。我会创建出函数来捕捉围绕这些数据的共用行为——可能只是一组共用的函数，也可能用一个类把数据结构与使用数据的函数组合起来。这个过程会改变代码的概念图景，将这些数据结构提升为新的抽象概念，可以帮助我更好地理解问题域。果真如此，这个重构过程会产生惊人强大的效用——但如果不用引入参数对象开启这个过程，后面的一切都不会发生。

做法

- 如果暂时还没有一个合适的数据结构，就创建一个。

我倾向于使用类，因为稍后把行为放进来会比较容易。我通常会尽量确保这些新建的数据结构是值对象[mf-vo]。

- 测试。
- 使用改变函数声明（124）给原来的函数新增一个参数，类型是新建的数据结构。
- 测试。
- 调整所有调用者，传入新数据结构的适当实例。每修改一处，执行测试。
- 用新数据结构中的每项元素，逐一取代参数列表中与之对应的参数项，然后删除原来的参数。测试。

范例

下面要展示的代码会查看一组温度读数

（reading），检查是否有任何一条读数超出了指定的运作温度范围（range）。温度读数的数据如下：

```
const station = { name: "ZB1",
                  readings: [
                    {temp: 47, time: "2016-11-10 09:10"},
                    {temp: 53, time: "2016-11-10 09:20"},
                    {temp: 58, time: "2016-11-10 09:30"},
                    {temp: 53, time: "2016-11-10 09:40"},
                    {temp: 51, time: "2016-11-10 09:50"},
                  ]
                };
```

下面的函数负责找到超出指定范围的温度读数：

```
function readingsOutsideRange(station, min, max) {
  return station.readings
    .filter(r => r.temp < min || r.temp > max);
}
```

调用该函数的代码可能是下面这样的。

调用方

```
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling);
```

请注意，这里的调用代码从另一个对象中抽出两项数据，转手又把这一对数据传递给readingsOutsideRange。代表“运作计划”的operatingPlan对象用了另外的名字来表示温度范围的下限和上限，与readingsOutsideRange中所用的名字不同。像这样用两项各不相干的数据来表示一个范围的情况并不少见，最好是将其组合成一个对象。我会首先为要组合的数据声明一个类：

```
class NumberRange {
  constructor(min, max) {
    this._data = {min: min, max: max};
  }
  get min() {return this._data.min;}
  get max() {return this._data.max;}
}
```

我声明了一个类，而不是基本的JavaScript对象，因为这个重构通常只是一系列重构的起点，随后我会把行为搬移到新建的对象中。既然类更适合承载数据与行为的组合，我就直接从声明一个类开始。同时，在这个新类中，我不会提供任何更新数据的函数，因为我有可能将其处理成值对象（Value Object）[mf-vo]。在使用这个重构手法时，大多数情况下我都会创建值对象。

然后我会运用改变函数声明（124），把新的对象作为参数传给readingsOutsideRange。

```
function readingsOutsideRange(station, min, max, range) {
  return station.readings
    .filter(r => r.temp < min || r.temp > max);
}
```

在JavaScript中，此时我不需要修改调用方代码，但在其他语言中，我必须在调用处为新参数传入`null`值，就像下面这样。

调用方

```
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              null);
```

到目前为止，我还没有修改任何行为，所以测试应该仍然能通过。随后，我会挨个找到函数的调用处，传入合适的温度范围。

调用方

```
const range = new NumberRange(operatingPlan.temperatureFloor,
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              range);
```

此时我还是没有修改任何行为，因为新添的参数没有被使用。所有测试应该仍然能通过。

现在我可以开始修改使用参数的代码了。先从“最大值”开始：

```
function readingsOutsideRange(station, min, max, range) {
  return station.readings
    .filter(r => r.temp < min || r.temp > range.max);
}
```

调用方

```
const range = new NumberRange(operatingPlan.temperatureFloor,
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              range);
```

此时要执行测试。如果测试通过，我再接着处理另一个参数。

```
function readingsOutsideRange(station, min, range) {
  return station.readings
    .filter(r => r.temp < range.min || r.temp > range.max);
}
```

调用方

```
const range = new NumberRange(operatingPlan.temperatureFloor,
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              range);
```

这项重构手法到这儿就完成了。不过，将一堆参数替换成一个真正的对象，这只是长征第一步。创建一个类是为了把行为搬移进去。在这里，我可以给“范围”类添加一个函数，用于测试一个值是否落在范围之内。

```
function readingsOutsideRange(station, range) {
  return station.readings
    .f ilter(r => !range.contains(r.temp));
}
```

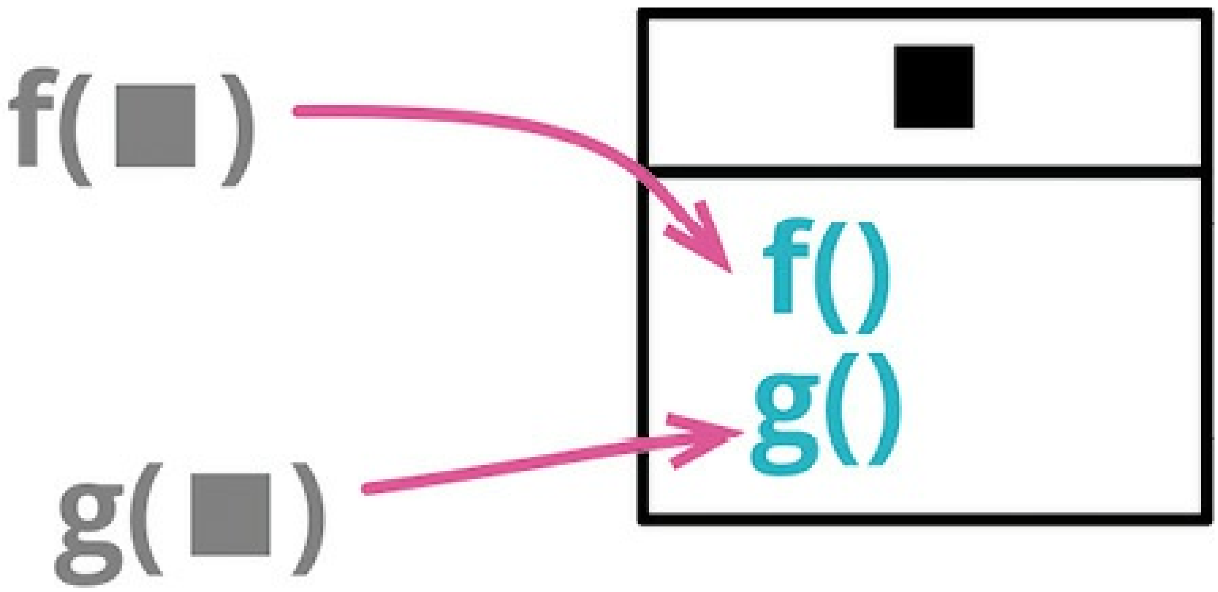
class NumberRange...

```
contains(arg) {return (arg >= this.min && arg <= this.max);}
```

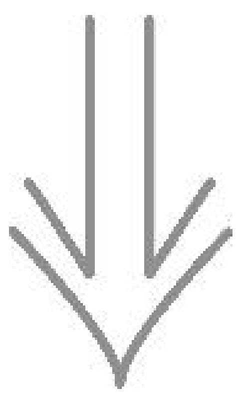
这样我就迈出了第一步，开始逐渐打造一个真正有用的“范围”[mf-range]类。一旦识别出“范围”这个概念，那么每当我在代码中发现“最大/最小值”这

样一对数字时，我就会考虑是否可以将其改为使用“范围”类。（例如，我马上就会考虑把“运作计划”类中的temperatureFloor和temperatureCeiling替换为temperatureRange。）在观察这些成对出现的数字如何被使用时，我会发现一些有用的行为，并将其搬移到“范围”类中，简化其使用方法。比如，我可能会先给这个类加上“基于数值判断相等性”的函数，使其成为一个真正的值对象。

6.9　函数组合成类（Combine Functions into Class）




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
```



```
class Reading {
  base() {...}
  taxableCharge() {...}
  calculateBaseCharge() {...}
}
```

动机

类，在大多数现代编程语言中都是基本的构造。它们把数据与函数捆绑到同一个环境中，将一部分数据与函数暴露给其他程序元素以便协作。它们是面向对象语言的首要构造，在其他程序设计方法中也同样有用。

如果发现一组函数形影不离地操作同一块数据（通常是将这块数据作为参数传递给函数），我就认为，是时候组建一个类了。类能明确地给这些函数提供一个共用的环境，在对象内部调用这些函数可以少传许多参数，从而简化函数调用，并且这样一个对象也可以更方便地传递给系统的其他部分。

除了可以把已有的函数组织起来，这个重构还给我们一个机会，去发现其他的计算逻辑，将它们也重构到新的类当中。

将函数组织到一起的另一种方式是函数组合成

变换（149）。具体使用哪个重构手法，要看程序整体的上下文。使用类有一大好处：客户端可以修改对象的核心数据，通过计算得出的派生数据则会自动与核心数据保持一致。

类似这样的一组函数不仅可以组合成一个类，而且可以组合成一个嵌套函数。通常我更倾向于类而非嵌套函数，因为后者测试起来会比较困难。如果我想对外暴露多个函数，也必须采用类的形式。

在有些编程语言中，类不是一等公民，而函数则是。面对这样的语言，可以用“函数作为对象”（Function As Object）[mf-fao]的形式来实现这个重构手法。

做法

- 运用封装记录（162）对多个函数共用的数据记录加以封装。

  如果多个函数共用的数据还未组织成记录结构，则先运用引入参数对象（140）将其组织成记录。

- 对于使用该记录结构的每个函数，运用搬移函数（198）将其移入新类。

如果函数调用时传入的参数已经是新类的成员，则从参数列表中去除之。

- 用以处理该数据记录的逻辑可以用提炼函数（106）提炼出来，并移入新类。

范例

我在英格兰长大，那是一个热爱喝茶的国度。（个人而言，我不喜欢在英格兰喝到的大部分茶，对中国茶和日本茶倒是情有独钟。）所以，我虚构了一种用于向老百姓供给茶水的公共设施。每个月会有软件读取茶水计量器的数据，得到类似这样的读数（reading）：

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 20
```

浏览处理这些数据记录的代码，我发现有很多地方在做着相似的计算，于是我找到了一处计算“基础费用”（base charge）的逻辑。

客户端1...

```
const aReading = acquireReading();
const baseCharge = baseRate(aReading.month, aReading.year) *
```

在英格兰，一切生活必需品都得交税，茶自然也不例外。不过，按照规定，只要不超出某个必要用量，就不用交税。

客户端2...

```
const aReading = acquireReading();
const base = (baseRate(aReading.month, aReading.year) * aRead
const taxableCharge = Math.max(0, base - taxThreshold(aReadin
```

我相信你也发现了：计算基础费用的公式被重复了两遍。如果你跟我有一样的习惯，现在大概已经在着手提炼函数（106）了。有趣的是，好像别人已经动过这个脑筋了。

客户端3...

```
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);

function calculateBaseCharge(aReading) {
  return baseRate(aReading.month, aReading.year) * aReading.q
}
```

看到这里，我有一种自然的冲动，想把前面两处客户端代码都改为使用这个函数。但这样一个顶层函数的问题在于，它通常位于一个文件中，读者不一定能想到来这里寻找它。我更愿意对代码多做些修改，让该函数与其处理的数据在空间上有更紧密的联系。为此目的，不妨把数据本身变成一个类。

我可以运用封装记录（162）将记录变成类。

```
class Reading {
  constructor(data) {
    this._customer = data.customer;
    this._quantity = data.quantity;
    this._month = data.month;
    this._year = data.year;
  }
  get customer() {return this._customer;}
  get quantity() {return this._quantity;}
  get month()    {return this._month;}
  get year()     {return this._year;}
}
```

首先，我想把手上已有的函数calculateBaseCharge搬到新建的Reading类中。一得到原始的读数数据，我就用Reading类将它包装起来，然后就可以在函数中使用Reading类了。

客户端3...

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

然后我用搬移函数（198）把calculateBaseCharge搬到新类中。

class Reading...

```
get calculateBaseCharge() {
  return baseRate(this.month, this.year) * this.quantity;
}
```

客户端3...

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.calculateBaseCharge;
```

搬移的同时，我会顺便运用函数改名（124），按照我喜欢的风格对这个函数改名。

```
get baseCharge() {
  return baseRate(this.month, this.year) * this.quantity;
}
```

客户端3...

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

用这个名字，`Reading`类的客户端将不知道`baseCharge`究竟是一个字段还是推演计算出的值。这是好事，它符合“统一访问原则”（Uniform Access Principle）[mf-ua]。

现在我可以修改客户端1的代码，令其调用新的方法，不要重复计算基础费用。

客户端1...

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const baseCharge = aReading.baseCharge;
```

很有可能我会顺手用内联变量（123）把baseCharge变量给去掉。不过，我们当下介绍的重构手法更关心“计算应税费用”的逻辑。同样，我先将那里的客户端代码改为使用新建的baseCharge属性。

客户端2...

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = Math.max(0, aReading.baseCharge - taxTh
```

运用提炼函数（106）将计算应税费用（taxable charge）的逻辑提炼成函数：

```
function taxableChargeFn(aReading) {
  return Math.max(0, aReading.baseCharge - taxThreshold(aRead
}
```

客户端3...

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = taxableChargeFn(aReading);
```

然后我运用搬移函数（198）将其移入Reading类：

class Reading...

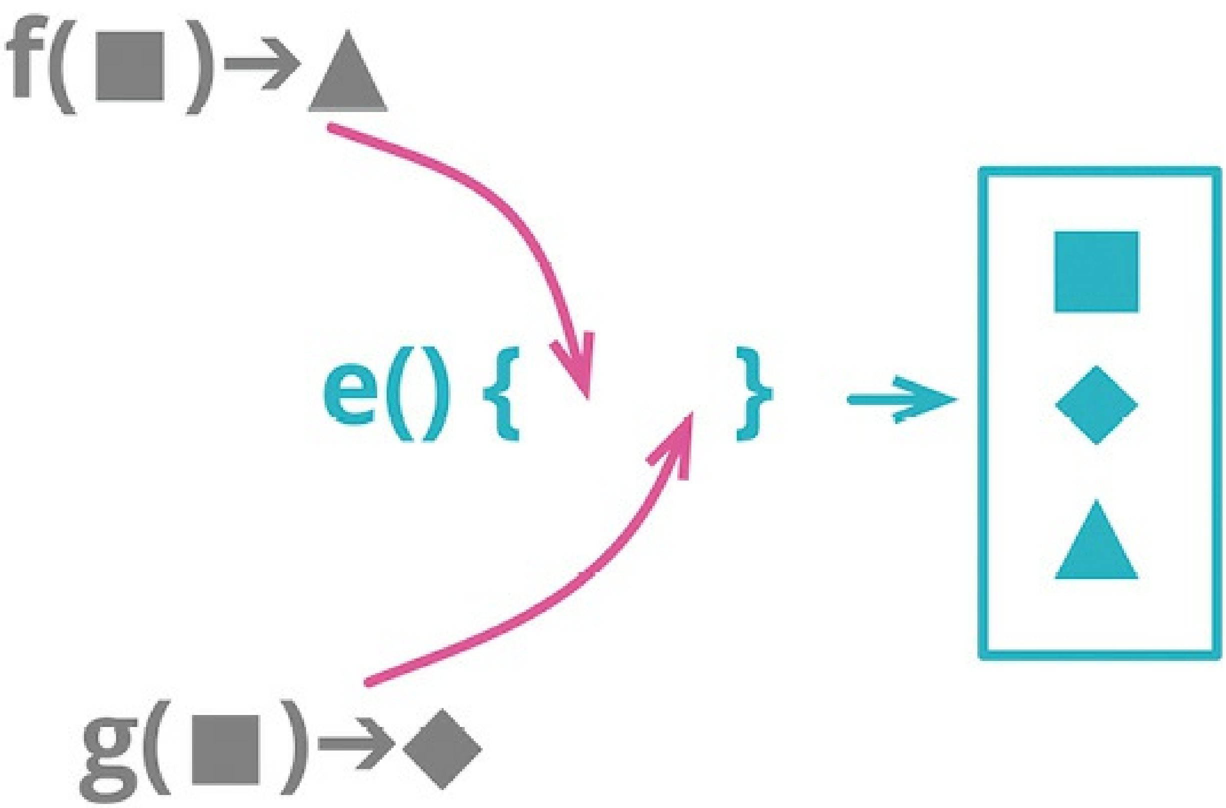
```
get taxableCharge() {
  return Math.max(0, this.baseCharge - taxThreshold(this.year
}
```

客户端3...

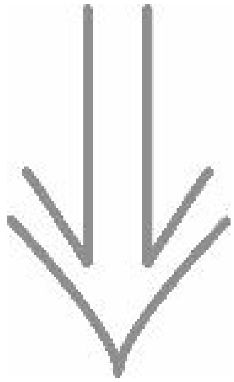
```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

由于所有派生数据都是在使用时计算得出的，所以对存储下来的读数进行修改也没问题。一般而论，我更倾向于使用不可变的数据；但很多时候我们必须得使用可变数据（比如JavaScript整个语言生态在设计时就没有考虑数据的不可变性）。如果数据确有可能被更新，那么用类将其封装起来会很有帮助。

6.10　函数组合成变换（Combine Functions into Transform）




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
```



```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading);
  aReading.baseCharge = base(aReading);
  aReading.taxableCharge = taxableCharge(aReading);
  return aReading;
}
```

动机

在软件中，经常需要把数据“喂”给一个程序，让它再计算出各种派生信息。这些派生数值可能会在几个不同地方用到，因此这些计算逻辑也常会在用到派生数据的地方重复。我更愿意把所有计算派生数据的逻辑收拢到一处，这样始终可以在固定的地方找到和更新这些逻辑，避免到处重复。

一个方式是采用数据变换（transform）函数：这种函数接受源数据作为输入，计算出所有的派生数据，将派生数据以字段形式填入输出数据。有了

变换函数，我就始终只需要到变换函数中去检查计算派生数据的逻辑。

函数组合成变换的替代方案是函数组合成类（144），后者的做法是先用源数据创建一个类，再把相关的计算逻辑搬移到类中。这两个重构手法都很有用，我常会根据代码库中已有的编程风格来选择使用其中哪一个。不过，两者有一个重要的区别：如果代码中会对源数据做更新，那么使用类要好得多；如果使用变换，派生数据会被存储在新生成的记录中，一旦源数据被修改，我就会遭遇数据不一致。

我喜欢把函数组合起来的原因之一，是为了避免计算派生数据的逻辑到处重复。从道理上来说，只用提炼函数（106）也能避免重复，但孤立存在的函数常常很难找到，只有把函数和它们操作的数据放在一起，用起来才方便。引入变换（或者类）都是为了让相关的逻辑找起来方便。

做法

- 创建一个变换函数，输入参数是需要变换的记录，并直接返回该记录的值。

这一步通常需要对输入的记录做深复制（deep copy）。此时应该写个测试，确保变换不会修改原来的记录。

- 挑选一块逻辑，将其主体移入变换函数中，把结果作为字段添加到输出记录中。修改客户端代码，令其使用这个新字段。

如果计算逻辑比较复杂，先用提炼函数（106）提炼之。

- 测试。
- 针对其他相关的计算逻辑，重复上述步骤。

范例

在我长大的国度，茶是生活中的重要部分，以至于我想象了这样一种特别的公共设施，专门给老百姓供应茶水。每个月，从这个设备上可以得到读数（reading），从而知道每位顾客取用了多少茶。

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 20
```

几个不同地方的代码分别根据茶的用量进行计算。一处是计算应该向顾客收取的基本费用。

客户端1...

```
const aReading = acquireReading();
const baseCharge = baseRate(aReading.month, aReading.year) *
```

另一处是计算应该交税的费用—比基本费用要少，因为政府明智地认为，每个市民都有权免税享受一定量的茶水。

客户端2...

```
const aReading = acquireReading();
const base = (baseRate(aReading.month, aReading.year) * aRead
const taxableCharge = Math.max(0, base - taxThreshold(aReadin
```

浏览处理这些数据记录的代码，我发现有很多地方在做着相似的计算。这样的重复代码，一旦需要修改（我打赌这只是早晚的问题），就会造成麻烦。我可以用提炼函数（106）来处理这些重复的

计算逻辑，但这样提炼出来的函数会散落在程序中，以后的程序员还是很难找到。说真的，我还真在另一块代码中找到了一个这样的函数。

客户端3...

```
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);

function calculateBaseCharge(aReading) {
  return baseRate(aReading.month, aReading.year) * aReading.q
}
```

处理这种情况的一个办法是，把所有这些计算派生数据的逻辑搬移到一个变换函数中，该函数接受原始的“读数”作为输入，输出则是增强的“读数”记录，其中包含所有共用的派生数据。

我先要创建一个变换函数，它要做的事很简单，就是复制输入的对象：

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  return result;
}
```

我用了Lodash库的`cloneDeep`函数来进行深复制。

这个变换函数返回的本质上仍是原来的对象，只是添加了更多的信息在上面。对于这种函数，我喜欢用“enrich”（增强）这个词来给它命名。如果它生成的是跟原来完全不同的对象，我就会用“transform”（变换）来命名它。

然后我挑选一处想要搬移的计算逻辑。首先，我用现在的`enrichReading`函数来增强“读数”记录，尽管该函数暂时还什么都没做。

客户端3...

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

然后我运用搬移函数（198）把`calculateBaseCharge`函数搬移到增强过程中：

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  return result;
}
```

在变换函数内部，我乐得直接修改结果对象，而不是每次都复制一个新对象。我喜欢不可变的数

据，但在大部分编程语言中，保持数据完全不可变很困难。在程序模块的边界处，我做好了心理准备，多花些精力来支持不可变性。但在较小的范围内，我可以接受可变的数据。另外，我把这里用到的变量命名为aReading，表示它是一个累积变量（accumulating variable）。这样当我把更多的逻辑搬移到变换函数enrichReading中时，这个变量名也仍然适用。

修改客户端代码，令其改用增强后的字段：

客户端3...

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

当所有调用calculateBaseCharge的地方都修改完成后，就可以把这个函数内嵌到enrichReading函数中，从而更清楚地表明态度：如果需要“计算基本费用”的逻辑，请使用增强后的记录。

在这里要当心一个陷阱：在编写enrichReading函数时，我让它返回了增强后的读数记录，这背后隐含的意思是原始的读数记录不会

被修改。所以我最好为此加个测试。

```
it('check reading unchanged', function() {
  const baseReading = {customer: "ivan", quantity: 15, month:
  const oracle = _.cloneDeep(baseReading);
  enrichReading(baseReading);
  assert.deepEqual(baseReading, oracle);
});
```

现在我可以修改客户端1的代码，让它也使用这个新添的字段。

客户端1...

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const baseCharge = aReading.baseCharge;
```

此时可以考虑用内联变量（123）去掉`baseCharge`变量。

现在我转头去看“计算应税费用”的逻辑。第一步是把变换函数用起来：

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base = (baseRate(aReading.month, aReading.year) * aRead
const taxableCharge = Math.max(0, base - taxThreshold(aReadin
```

基本费用的计算逻辑马上就可以改用变换得到的新字段代替。如果计算逻辑比较复杂，我可以先运用提炼函数（106）。不过这里的情况足够简单，一步到位修改过来就行。

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base = aReading.baseCharge;
const taxableCharge = Math.max(0, base - taxThreshold(aReadin
```

执行测试之后，我就用内联变量（123）去掉base变量：

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const taxableCharge = Math.max(0, aReading.baseCharge - taxTh
```

然后把计算逻辑搬移到变换函数中：

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  result.taxableCharge = Math.max(0, result.baseCharge - taxT
  return result;
}
```

修改使用方代码，让它使用新添的字段。

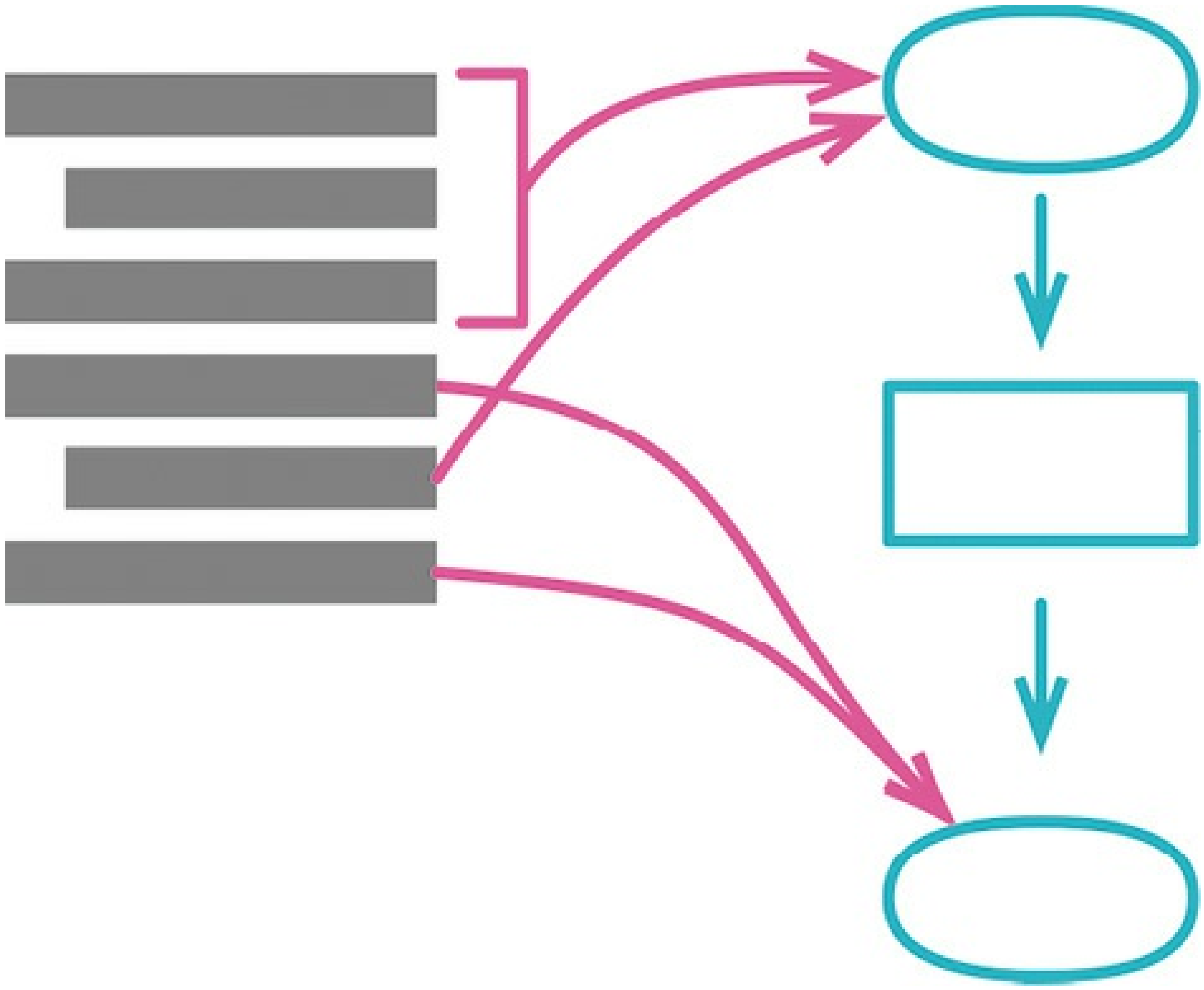
```
const rawReading = acquireReading();
```

```
const aReading = enrichReading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

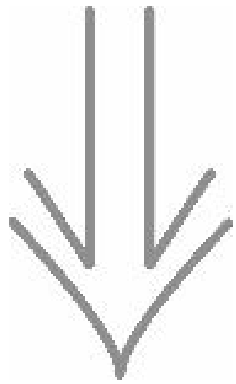
测试。现在我可以再次用内联变量（123）把taxableCharge变量也去掉。

增强后的读数记录有一个大问题：如果某个客户端修改了一项数据的值，会发生什么？比如说，如果某处代码修改了quantity字段的值，就会导致数据不一致。在JavaScript中，避免这种情况最好的办法是不要使用本重构手法，改用函数组合成类（144）。如果编程语言支持不可变的数据结构，那么就没有这个问题了，那样的语言中会更常用到变换。但即便编程语言不支持数据结构不可变，如果数据是在只读的上下文中被使用（例如在网页上显示派生数据），还是可以使用变换。

6.11　拆分阶段（Split Phase）



```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```



```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```

动机

每当看见一段代码在同时处理两件不同的事，我就想把它拆分成各自独立的模块，因为这样到了需要修改的时候，我就可以单独处理每个主题，而不必同时在脑子里考虑两个不同的主题。如果运气够好的话，我可能只需要修改其中一个模块，完全不用回忆起另一个模块的诸般细节。

最简洁的拆分方法之一，就是把一大段行为分成顺序执行的两个阶段。可能你有一段处理逻辑，其输入数据的格式不符合计算逻辑的要求，所以你得先对输入数据做一番调整，使其便于处理。也可能是你把数据处理逻辑分成顺序执行的多个步骤，每个步骤负责的任务全然不同。

编译器是最典型的例子。编译器的任务很直观：接受文本（用某种编程语言编写的代码）作为输入，将其转换成某种可执行的格式（例如针对某种特定硬件的目标码）。随着经验加深，我们发现把这项大任务拆分成一系列阶段会很有帮助：首先对文本做词法分析，然后把token解析成语法树，然后再对语法树做几步转换（如优化），最后生成目标码。每一步都有边界明确的范围，我可以聚焦思考其中一步，而不用理解其他步骤的细节。

在大型软件中，类似这样的阶段拆分很常见，例如编译器的每个阶段又包含若干函数和类。即便只有不大的一块代码，只要我发现了有益的将其拆分成多个阶段的机会，同样可以运用拆分阶段重构手法。如果一块代码中出现了上下几段，各自使用不同的一组数据和函数，这就是最明显的线索。将这些代码片段拆分成各自独立的模块，能更明确地标示出它们之间的差异。

做法

- 将第二阶段的代码提炼成独立的函数。
- 测试。
- 引入一个中转数据结构，将其作为参数添加到提炼出的新函数的参数列表中。
- 测试。
- 逐一检查提炼出的“第二阶段函数”的每个参数。如果某个参数被第一阶段用到，就将其移入中转数据结构。每次搬移之后都要执行测试。

　　有时第二阶段根本不应该使用某个参数。果真如此，就把使用该参数得到的结果全都提炼成中转数据结构的字段，然后用搬移语句到调用者（217）把使用该参数的代码行搬移到“第二阶段函数”之外。

- 对第一阶段的代码运用提炼函数（106），让提炼出的函数返回中转数据结构。

也可以把第一阶段提炼成一个变换（transform）对象。

范例

我手上有一段“计算订单价格”的代码，至于订单中的商品是什么，我们从代码中看不出来，也不太关心。

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
          * product.basePrice * product.discountRate;

const shippingPerCase = (basePrice > shippingMethod.discountT
          ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

虽然只是个常见的、过于简单的范例，从中还是能看出有两个不同阶段存在的。前两行代码根据商品（product）信息计算订单中与商品相关的价格，随后的两行则根据配送（shipping）信息计算配送成本。后续的修改可能还会使价格和配送的计算逻辑变复杂，但只要这两块逻辑相对独立，将这

段代码拆分成两个阶段就是有价值的。

我首先用提炼函数（106）把计算配送成本的逻辑提炼出来。

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
        * product.basePrice * product.discountRate;

const price = applyShipping(basePrice, shippingMethod, quanti
  return price;
}
function applyShipping(basePrice, shippingMethod, quantity, d

const shippingPerCase = (basePrice > shippingMethod.discountT
        ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

第二阶段需要的数据都以参数形式传入。在真实环境下，参数的数量可能会很多，但我对此并不担心，因为很快就会将这些参数消除掉。

随后我会引入一个中转数据结构，使其在两阶段之间沟通信息。

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
        * product.basePrice * product.discountRate;
  const priceData = {};
```

```
const price = applyShipping(priceData, basePrice, shippingMet
  return price;
}

function applyShipping(priceData, basePrice, shippingMethod,

const shippingPerCase = (basePrice > shippingMethod.discountT
        ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

现在我会审视applyShipping的各个参数。第一个参数basePrice是在第一阶段代码中创建的，因此我将其移入中转数据结构，并将其从参数列表中去掉。

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
        * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice};

const price = applyShipping(priceData, basePrice, shippingMet
  return price;
}
function applyShipping(priceData, basePrice, shippingMethod,

const shippingPerCase = (priceData.basePrice > shippingMethod
        ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = quantity * shippingPerCase;

const price = priceData.basePrice - discount + shippingCost;
  return price;
}
```

下一个参数是shippingMethod。第一阶段中没有使用这项数据，所以它可以保留原样。

再下一个参数是quantity。这个参数在第一阶段中用到，但不是在那里创建的，所以其实可以将其留在参数列表中。但我更倾向于把尽可能多的参数搬移到中转数据结构中。

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
        * product.basePrice * product.discountRate;

const priceData = {basePrice: basePrice, quantity: quantity};

const price = applyShipping(priceData, shippingMethod, quanti
  return price;
}
function applyShipping(priceData, shippingMethod, quantity, d

const shippingPerCase = (priceData.basePrice > shippingMethod
        ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = priceData.quantity * shippingPerCase;

const price = priceData.basePrice - discount + shippingCost;
  return price;
}
```

对discount参数我也如法炮制。

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
        * product.basePrice * product.discountRate;
```

```
const priceData = {basePrice: basePrice, quantity: quantity,

const price = applyShipping(priceData, shippingMethod, discou
  return price;
}
function applyShipping(priceData, shippingMethod, discount) {

const shippingPerCase = (priceData.basePrice > shippingMethod
        ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = priceData.quantity * shippingPerCase;

const price = priceData.basePrice - priceData.discount + ship
  return price;
}
```

处理完参数列表后，中转数据结构得到了完整的填充，现在我可以把第一阶段代码提炼成独立的函数，令其返回这份数据。

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  const price = applyShipping(priceData, shippingMethod);
  return price;
}
function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
        * product.basePrice * product.discountRate;

return {basePrice: basePrice, quantity: quantity, discount:di
}
function applyShipping(priceData, shippingMethod) {

const shippingPerCase = (priceData.basePrice > shippingMethod
        ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = priceData.quantity * shippingPerCase;

const price = priceData.basePrice - priceData.discount + ship
  return price;
}
```

两个函数中，最后一个const变量都是多余的，我忍不住洁癖，将它们内联消除掉。

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  return applyShipping(priceData, shippingMethod);
}


function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;

const discount = Math.max(quantity - product.discountThreshol
        * product.basePrice * product.discountRate;

return {basePrice: basePrice, quantity: quantity, discount:di
}
function applyShipping(priceData, shippingMethod) {

const shippingPerCase = (priceData.basePrice > shippingMethod
        ? shippingMethod.discountedFee : shippingMethod.feePe
  const shippingCost = priceData.quantity * shippingPerCase;

return priceData.basePrice - priceData.discount + shippingCos
}
```

第7章　封装

分解模块时最重要的标准，也许就是识别出那些模块应该对外界隐藏的小秘密了[Parnas]。数据结构无疑是最常见的一种秘密，我可以用封装记录（162）或封装集合（170）手法来隐藏它们的细节。即便是基本类型的数据，也能通过以对象取代基本类型（174）进行封装——这样做后续所带来的巨大收益通常令人惊喜。另一项经常在重构时挡道的是临时变量，我需要确保它们的计算次序正确，还得保证其他需要它们的地方能获得其值。这里以查询取代临时变量（178）手法可以帮上大忙，特别是在分解一个过长的函数时。

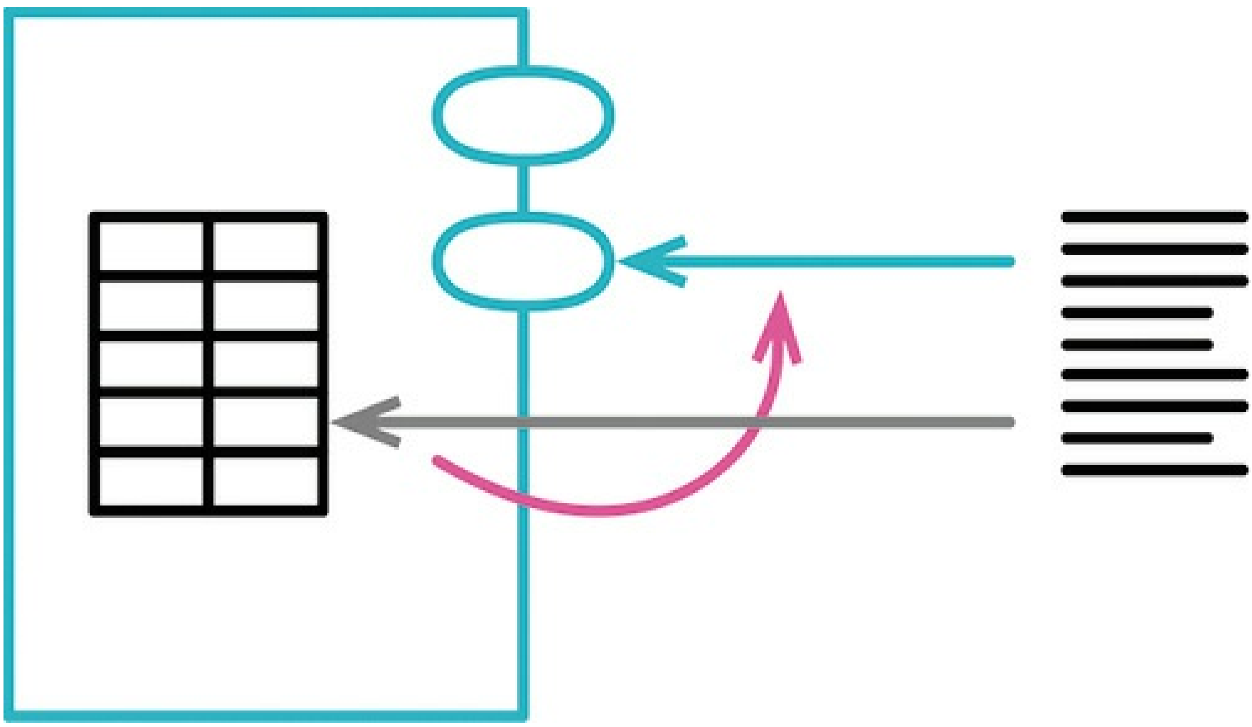
类是为隐藏信息而生的。在第6章中，我已经介绍了使用函数组合成类（144）手法来形成类的办法。此外，一般的提炼/内联操作对类也适用，见提炼类（182）和内联类（186）。

除了类的内部细节，使用隐藏委托关系（189）隐藏类之间的关联关系通常也很有帮助。但过多隐藏也会导致冗余的中间接口，此时我就需要它的反向重构——移除中间人（192）。

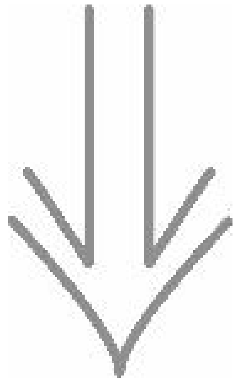
类与模块已然是施行封装的最大实体了，但小一点的函数对于封装实现细节也有所裨益。有时候，我可能需要将一个算法完全替换掉，这时我可以用提炼函数（106）将算法包装到函数中，然后使用替换算法（195）。

7.1　封装记录（Encapsulate Record）

曾用名：以数据类取代记录（Replace Record with Data Class）



```
organization = {name: "Acme Gooseberries", country: "GB"};
```



```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(arg) {this._name = arg;}
  get country()    {return this._country;}
  set country(arg) {this._country = arg;}
}
```

动机

记录型结构是多数编程语言提供的一种常见特性。它们能直观地组织起存在关联的数据，让我可以将数据作为有意义的单元传递，而不仅是一堆数据的拼凑。但简单的记录型结构也有缺陷，最恼人的一点是，它强迫我清晰地区分“记录中存储的数据”和“通过计算得到的数据”。假使我要描述一个整数闭区间，我可以用`{start: 1, end: 5}`描述，

或者用{start: 1, length: 5}（甚至还能用{end: 5, length: 5}，如果我想露两手华丽的编程技巧的话）。但不论如何存储，这3个值都是我想知道的，即区间的起点（start）和终点（end），以及区间的长度（length）。

这就是对于可变数据，我总是更偏爱使用类对象而非记录的原因。对象可以隐藏结构的细节，仅为这3个值提供对应的方法。该对象的用户不必追究存储的细节和计算的过程。同时，这种封装还有助于字段的改名：我可以重新命名字段，但同时提供新老字段名的访问方法，这样我就可以渐进地修改调用方，直到替换全部完成。

注意，我所说的偏爱对象，是对可变数据而言。如果数据不可变，我大可直接将这3个值保存在记录里，需要做数据变换时增加一个填充步骤即可。重命名记录也一样简单，你可以复制一个字段并逐步替换引用点。

记录型结构可以有两种类型：一种需要声明合法的字段名字，另一种可以随便用任何字段名字。后者常由语言库本身实现，并通过类的形式提供出来，这些类称为散列（hash）、映射（map）、散列映射（hashmap）、字典（dictionary）或关联数组（associative array）等。很多编程语言都提供了方便的语法来创建这类记录，这使得它们在各种编

程场景下都能大展身手。但使用这类结构也有缺陷，那就是一条记录上持有什么字段往往不够直观。比如说，如果我想知道记录里维护的字段究竟是起点/终点还是起点/长度，就只有查看它的创建点和使用点，除此以外别无他法。若这种记录只在程序的一个小范围里使用，那问题还不大，但若其使用范围变宽，“数据结构不直观”这个问题就会造成更多困扰。我可以重构它，使其变得更直观——但如果真需要这样做，那还不如使用类来得直接。

程序中间常常需要互相传递嵌套的列表（list）或散列映射结构，这些数据结构后续经常需要被序列化成JSON或XML。这样的嵌套结构同样值得封装，这样，如果后续其结构需要变更或者需要修改记录内的值，封装能够帮我更好地应对变化。

做法

- 对持有记录的变量使用封装变量（132），将其封装到一个函数中。

记得为这个函数取一个容易搜索的名字。

- 创建一个类，将记录包装起来，并将记录变量的值替换为该类的一个实例。然后在类上定义一个访问函数，用于返回原始的记录。修改封装变量的函数，令其使用这个访问函数。
- 测试。
- 新建一个函数，让它返回该类的对象，而非那条原始的记录。
- 对于该记录的每处使用点，将原先返回记录的函数调用替换为那个返回实例对象的函数调用。使用对象上的访问函数来获取数据的字段，如果该字段的访问函数还不存在，那就创建一个。每次更改之后运行测试。

如果该记录比较复杂，例如是个嵌套解构，那么先重点关注客户端对数据的更新操作，对于读取操作可以考虑返回一个数据副本或只读的数据代理。

- 移除类对原始记录的访问函数，那个容易搜索的返回原始数据的函数也要一并删除。
- 测试。

- 如果记录中的字段本身也是复杂结构，考虑对其再次应用封装记录（162）或封装集合（170）手法。

范例

首先，我从一个常量开始，该常量在程序中被大量使用。

```
const organization = {name: "Acme Gooseberries", country: "GB
```

这是一个普通的JavaScript对象，程序中很多地方都把它当作记录型结构在使用。以下是对其进行读取和更新的地方：

```
result += `<h1>${organization.name}</h1>`;
organization.name = newName;
```

重构的第一步很简单，先施展一下封装变量（132）。

```
function getRawDataOfOrganization() {return organization;}
```

读取的例子...

```
result += `<h1>${getRawDataOfOrganization().name}</h1>`;
```

更新的例子...

```
getRawDataOfOrganization().name = newName;
```

这里施展的不全是标准的封装变量（132）手法，我刻意为设值函数取了一个又丑又长、容易搜索的名字，因为我有意不让它在这次重构中活得太久。

封装记录意味着，仅仅替换变量还不够，我还想控制它的使用方式。我可以用类来替换记录，从而达到这一目的。

class Organization...

```
class Organization {
  constructor(data) {
    this._data = data;
  }
}
```

顶层作用域

```
const organization = new Organization({name: "Acme Gooseberri

function getRawDataOfOrganization() {return organization._dat
function getOrganization() {return organization;}
```

创建完对象后，我就能开始寻找该记录的使用点了。所有更新记录的地方，用一个设值函数来替换它。

class Organization...

```
set name(aString) {this._data.name = aString;}
```

客户端...

```
getOrganization().name = newName;
```

同样地，我将所有读取记录的地方，用一个取值函数来替代。

class Organization...

```
get name() {return this._data.name;}
```

客户端...

```
result += `<h1>${getOrganization().name}</h1>`;
```

完成引用点的替换后，就可以兑现我之前的死亡威胁，为那个名称丑陋的函数送终了。

```
function getRawDataOfOrganization() {return organization._dat
function getOrganization() {return organization;}
```

我还倾向于把_data里的字段展开到对象中。

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(aString) {this._name = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

这样做有一个好处，能够使外界无须再引用原始的数据记录。直接持有原始的记录会破坏封装的完整性。但有时也可能不适合将对象展开到独立的字段里，此时我就会先将_data复制一份，再进行赋值。

范例：封装嵌套记录

上面的例子将记录的浅复制展开到了对象里，但当我处理深层嵌套的数据（比如来自JSON文件的数据）时，又该怎么办呢？此时该重构手法的核心步骤依然适用，记录的更新点需要同样小心处理，但对记录的读取点则有多种处理方案。

作为例子，这里有一个嵌套层级更深的数据：它是一组顾客信息的集合，保存在散列映射中，并通过顾客ID进行索引。

```
"1920": {
  name: "martin",
  id: "1920",
  usages: {
    "2016": {
      "1": 50,
      "2": 55,
      // remaining months of the year
    },
    "2015": {
      "1": 70,
```

```
      "2": 63,
      // remaining months of the year
    }
  }
},
"38673": {
  name: "neal",
  id: "38673",
  // more customers in a similar form
```

对嵌套数据的更新和读取可以进到更深的层级。

更新的例子...

```
customerData[customerID].usages[year][month] = amount;
```

读取的例子...

```
function compareUsage (customerID, laterYear, month) {
  const later = customerData[customerID].usages[laterYear]
[month];

const earlier = customerData[customerID].usages[laterYear - 1
[month];
  return {laterAmount: later, change: later - earlier};
}
```

对这样的数据施行封装，第一步仍是封装变量

（132）。

```
function getRawDataOfCustomers() {return customerData;}
function setRawDataOfCustomers(arg) {customerData = arg;}
```

更新的例子...

```
getRawDataOfCustomers()[customerID].usages[year]
[month] = amount;
```

读取的例子...

```
function compareUsage (customerID, laterYear, month) {
           const      later      =      getRawDataOfCustomers()
[customerID].usages[laterYear][month];
          const     earlier     =     getRawDataOfCustomers()
[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

接下来我要创建一个类来容纳整个数据结构。

```
class CustomerData {
  constructor(data) {
    this._data = data;
  }
}
```

顶层作用域...

```
function getCustomerData() {return customerData;}
function getRawDataOfCustomers() {return customerData._data;}
function setRawDataOfCustomers(arg) {customerData = new Custo
```

最重要的是妥善处理好那些更新操作。因此，当我查看getRawDataOfCustomers的所有调用者时，总是特别关注那些对数据做修改的地方。再提醒你一下，下面是那步更新操作。

更新的例子...

```
getRawDataOfCustomers()[customerID].usages[year]
[month] = amount;
```

“做法”部分说，接下来要通过一个访问函数来返回原始的顾客数据，如果访问函数还不存在就创建一个。现在顾客类还没有设值函数，而且这个更新操作对结构进行了深入查找，因此是时候创建一个设值函数了。我会先用提炼函数（106），将层层深入数据结构的查找操作提炼到函数里。

更新的例子...

```
setUsage(customerID, year, month, amount);
```

顶层作用域...

```
function setUsage(customerID, year, month, amount) {
             getRawDataOfCustomers()[customerID].usages[year]
[month] = amount;
}
```

然后我再用搬移函数（198）将新函数搬移到新的顾客数据类中。

更新的例子...

```
getCustomerData().setUsage(customerID, year, month, amount);
```

class CustomerData...

```
setUsage(customerID, year, month, amount) {
  this._data[customerID].usages[year][month] = amount;
}
```

封装大型的数据结构时，我会更多关注更新操

作。凸显更新操作，并将它们集中到一处地方，是此次封装过程最重要的一部分。

一通替换过后，我可能认为修改已经告一段落，但如何确认替换是否真正完成了呢？检查的办法有很多，比如可以修改getRawDataOfCustomers函数，让其返回一份数据的深复制的副本。如果测试覆盖足够全面，那么当我真的遗漏了一些更新点时，测试就会报错。

顶层作用域...

```
function getCustomerData() {return customerData;}
function getRawDataOfCustomers() {return customerData.rawData
function setRawDataOfCustomers(arg) {customerData = new Custo
```

class CustomerData...

```
get rawData() {
  return _.cloneDeep(this._data);
}
```

我使用了lodash库来辅助生成深复制的副本。

另一个方式是，返回一份只读的数据代理。如

果客户端代码尝试修改对象的结构，那么该数据代理就会抛出异常。这在有些编程语言中能轻易实现，但用JavaScript实现可就麻烦了，我把它留给读者作为练习好了。或者，我可以复制一份数据，递归冻结副本的每个字段，以此阻止对它的任何修改企图。

妥善处理好数据的更新当然价值不凡，但读取操作又怎么处理呢？这有几种选择。

第一种选择是与设值函数采用同等待遇，把所有对数据的读取提炼成函数，并将它们搬移到CustomerData类中。

class CustomerData...

```
usage(customerID, year, month) {
  return this._data[customerID].usages[year][month];
}
```

顶层作用域**...**

```
function compareUsage (customerID, laterYear, month) {
  const later = getCustomerData().usage(customerID, laterYear
  const earlier = getCustomerData().usage(customerID, laterYe
  return {laterAmount: later, change: later - earlier};
}
```

这种处理方式的美妙之处在于，它为customerData提供了一份清晰的API列表，清楚描绘了该类的全部用途。我只需阅读类的代码，就能知道数据的所有用法。但这样会使代码量剧增，特别是当对象有许多用途时。现代编程语言大多提供直观的语法，以支持从深层的列表和散列[mf-lh]结构中获得数据，因此直接把这样的数据结构给到客户端，也不失为一种选择。

如果客户端想拿到一份数据结构，我大可以直接将实际的数据交出去。但这样做的问题在于，我将无从阻止用户直接对数据进行修改，进而使我们封装所有更新操作的良苦用心失去意义。最简单的应对办法是返回原始数据的一份副本，这可以用到我前面写的rawData方法。

class CustomerData...

```
get rawData() {
  return _.cloneDeep(this._data);
}
```

顶层作用域**...**

```
function compareUsage (customerID, laterYear, month) {
```

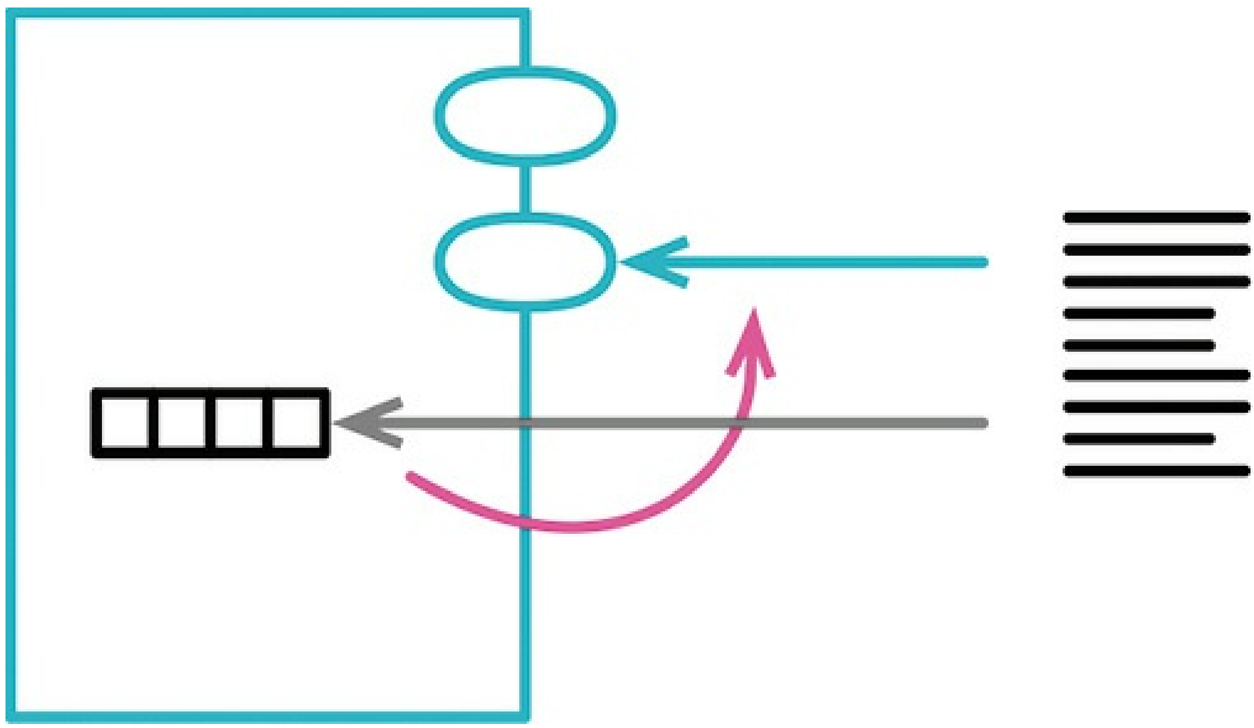
```
  const later = getCustomerData().rawData[customerID].usages[
[month];
  const earlier = getCustomerData().rawData[customerID].usage
[month];
  return {laterAmount: later, change: later - earlier};
}
```

简单归简单，这种方案也有缺点。最明显的问题是复制巨大的数据结构时代价颇高，这可能引发性能问题。不过也正如我对性能问题的一贯态度，这样的性能损耗也许是可以接受的——只有测量到可见的影响，我才会真的关心它。这种方案还可能带来困惑，比如客户端可能期望对该数据的修改会同时反映到原数据上。如果采用了只读代理或冻结副本数据的方案，就可以在此时提供一个有意义的错误信息。

另一种方案需要更多工作，但能提供更可靠的控制粒度：对每个字段循环应用封装记录。我会把顾客（customer）记录变成一个类，对其用途（usage）字段应用封装集合（170），并为它创建一个类。然后我就能通过访问函数来控制其更新点，比如说对用途（usage）对象应用将引用对象改为值对象（252）。但处理一个大型的数据结构时，这种方案异常繁复，如果对该数据结构的更新点没那么多，其实大可不必这么做。有时，合理混用取值函数和新对象可能更明智，即使用取值函数来封装数据的深层查找操作，但更新数据时则用对

象来包装其结构，而非直接操作未经封装的数据。我在“Refactoring Code to Load a Document”[mf-ref-doc]这篇文章中讨论了更多的细节，有兴趣的读者可移步阅读。

7.2　封装集合（Encapsulate Collection）



```
class Person {
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}
```

```
class Person {
  get courses() {return this._courses.slice();}
  addCourse(aCourse) { ... }
  removeCourse(aCourse) { ... }
```

动机

我喜欢封装程序中的所有可变数据。这使我很容易看清楚数据被修改的地点和修改方式，这样当我需要更改数据结构时就非常方便。我们通常鼓励封装——使用面向对象技术的开发者对封装尤为重视——但封装集合时人们常常犯一个错误：只对集合变量的访问进行了封装，但依然让取值函数返回集合本身。这使得集合的成员变量可以直接被修改，而封装它的类则全然不知，无法介入。

为避免此种情况，我会在类上提供一些修改集合的方法——通常是“添加”和“移除”方法。这样就

可使对集合的修改必须经过类，当程序演化变大时，我依然能轻易找出修改点。

只要团队拥有良好的习惯，就不会在模块以外修改集合，仅仅提供这些修改方法似乎也就足够。然而，依赖于别人的好习惯是不明智的，一个细小的疏忽就可能带来难以调试的bug。更好的做法是，不要让集合的取值函数返回原始集合，这就避免了客户端的意外修改。

一种避免直接修改集合的方法是，永远不直接返回集合的值。这种方法提倡，不要直接使用集合的字段，而是通过定义类上的方法来代替，比如将`aCustomer.orders.size`替换为`aCustomer.numberOfOrders`。我不同意这种做法。现代编程语言都提供了丰富的集合类和标准接口，能够组合成很多有价值的用法，比如集合管道（Collection Pipeline）[mf-cp]等。使用特殊的类方法来处理这些场景，会增加许多额外代码，使集合操作容易组合的特性大打折扣。

还有一种方法是，以某种形式限制集合的访问权，只允许对集合进行读操作。比如，在Java中可以很容易地返回集合的一个只读代理，这种代理允许用户读取集合，但会阻止所有更改操作——Java的代理会抛出一个异常。有一些库在构造集合时也用了类似的方法，将构造出的集合建立在迭代器或

枚举对象的基础上，因为迭代器也不能修改它迭代的集合。

也许最常见的做法是，为集合提供一个取值函数，但令其返回一个集合的副本。这样即使有人修改了副本，被封装的集合也不会受到影响。这可能带来一些困惑，特别是对那些已经习惯于通过修改返回值来修改原集合的开发者——但更多的情况下，开发者已经习惯于取值函数返回副本的做法。如果集合很大，这个做法可能带来性能问题，好在多数列表都没有那么大，此时前述的性能优化基本守则依然适用（见2.8节）。

使用数据代理和数据复制的另一个区别是，对源数据的修改会反映到代理上，但不会反映到副本上。大多数时候这个区别影响不大，因为通过此种方式访问的列表通常生命周期都不长。

采用哪种方法并无定式，最重要的是在同个代码库中做法要保持一致。我建议只用一种方案，这样每个人都能很快习惯它，并在每次调用集合的访问函数时期望相同的行为。

做法

- 如果集合的引用尚未被封装起来，先用封装变量（132）封装它。
- 在类上添加用于“添加集合元素”和“移除集合元素”的函数。

如果存在对该集合的设值函数，尽可能先用移除设值函数（331）移除它。如果不能移除该设值函数，至少让它返回集合的一份副本。

- 执行静态检查。
- 查找集合的引用点。如果有调用者直接修改集合，令该处调用使用新的添加/移除元素的函数。每次修改后执行测试。
- 修改集合的取值函数，使其返回一份只读的数据，可以使用只读代理或数据副本。
- 测试。

范例

假设有个人（`Person`）要去上课。我们用一个简单的`Course`来表示“课程”。

class Person...

```
constructor (name) {
  this._name = name;
  this._courses = [];
}
get name() {return this._name;}
get courses() {return this._courses;}
set courses(aList) {this._courses = aList;}
```

class Course...

```
constructor(name, isAdvanced) {
  this._name = name;
  this._isAdvanced = isAdvanced;
}
get name() {return this._name;}
get isAdvanced() {return this._isAdvanced;}
```

客户端会使用课程集合来获取课程的相关信息。

```
numAdvancedCourses = aPerson.courses
  .f ilter(c =&gt; c.isAdvanced)
  .length
;
```

有些开发者可能觉得这个类已经得到了恰当的封装，毕竟，所有的字段都被访问函数保护到了。

但我要指出，对课程列表的封装还不完整。诚然，对列表整体的任何更新操作，都能通过设值函数得到控制。

客户端代码...

```
const basicCourseNames = readBasicCourseNames(filename);
aPerson.courses = basicCourseNames.map(name => new Course(nam
```

但客户端也可能发现，直接更新课程列表显然更容易。

客户端代码...

```
for(const name of readBasicCourseNames(filename)) {
  aPerson.courses.push(new Course(name, false));
}
```

这就破坏了封装性，因为以此种方式更新列表Person类根本无从得知。这里仅仅封装了字段引用，而未真正封装字段的内容。

现在我来对类实施真正恰当的封装，首先要为类添加两个方法，为客户端提供“添加课程”和“移除课程”的接口。

class Person...

```
addCourse(aCourse) {
  this._courses.push(aCourse);
}
removeCourse(aCourse, fnIfAbsent = () => {throw new RangeErro
  const index = this._courses.indexOf(aCourse);
  if (index === -1) fnIfAbsent();
  else this._courses.splice(index, 1);
}
```

对于移除操作，我得考虑一下，如果客户端要求移除一个不存在的集合元素怎么办。我可以耸耸肩装作没看见，也可以抛出错误。这里我默认让它抛出错误，但留给客户端一个自己处理的机会。

然后我就可以让直接修改集合值的地方改用新的方法了。

客户端代码...

```
for(const name of readBasicCourseNames(filename)) {
  aPerson.addCourse(new Course(name, false));
}
```

有了单独的添加和移除方法，通常`setCourse`设值函数就没必要存在了。若果真如此，我就会使用移除设值函数（331）移除它。如果出于其他原

因，必须提供一个设值方法作为API，我至少要确保用一份副本给字段赋值，不去修改通过参数传入的集合。

class Person...

```
set courses(aList) {this._courses = aList.slice();}
```

这套设施让客户端能够使用正确的修改方法，同时我还希望能确保所有修改都通过这些方法进行。为达此目的，我会让取值函数返回一份副本。

class Person...

```
get courses() {return this._courses.slice();}
```

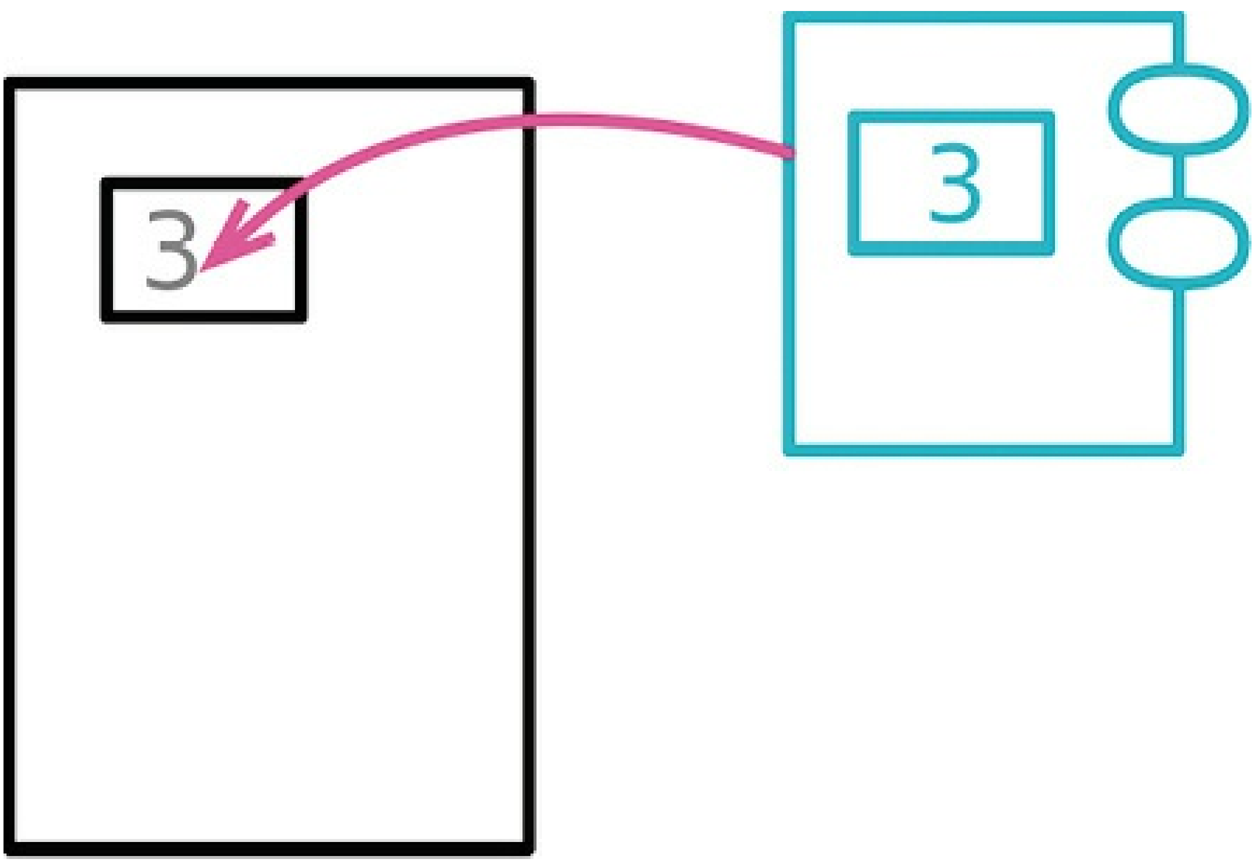
总的来讲，我觉得对集合保持适度的审慎是有益的，我宁愿多复制一份数据，也不愿去调试因意外修改集合招致的错误。修改操作并不总是显而易见的，比如，在JavaScript中原生的数组排序函数`sort()`就会修改原数组，而在其他语言中默认都是为更改集合的操作返回一份副本。任何负责管理集合的类都应该总是返回数据副本，但我还养成了

一个习惯，只要我做的事看起来可能改变集合，我也会返回一个副本。

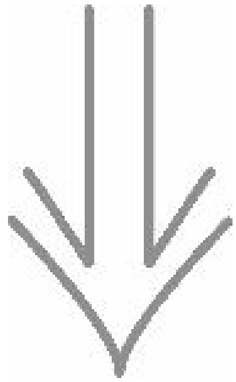
7.3　以对象取代基本类型（Replace Primitive with Object）

曾用名：以对象取代数据值（Replace Data Value with Object）

曾用名：以类取代类型码（Replace Type Code with Class）




```
orders.filter(o => "high" === o.priority
                || "rush" === o.priority);
```



```
orders.filter(o => o.priority.higherThan(new Priority("normal
```

动机

开发初期，你往往决定以简单的数据项表示简单的情况，比如使用数字或字符串等。但随着开发的进行，你可能会发现，这些简单数据项不再那么简单了。比如说，一开始你可能会用一个字符串来表示“电话号码”的概念，但是随后它又需要“格式化”“抽取区号”之类的特殊行为。这类逻辑很快便会占领代码库，制造出许多重复代码，增加使用时的成本。

一旦我发现对某个数据的操作不仅仅局限于打印时，我就会为它创建一个新类。一开始这个类也许只是简单包装一下简单类型的数据，不过只要类有了，日后添加的业务逻辑就有地可去了。这些小

小的封装值开始可能价值甚微，但只要悉心照料，它们很快便能成长为有用的工具。创建新类无须太大的工作量，但我发现它们往往对代码库有深远的影响。实际上，许多经验丰富的开发者认为，这是他们的工具箱里最实用的重构手法之一——尽管其价值常为新手程序员所低估。

做法

- 如果变量尚未被封装起来，先使用封装变量（132）封装它。
- 为这个数据值创建一个简单的类。类的构造函数应该保存这个数据值，并为它提供一个取值函数。
- 执行静态检查。
- 修改第一步得到的设值函数，令其创建一个新类的对象并将其存入字段，如果有必要的话，同时修改字段的类型声明。
- 修改取值函数，令其调用新类的取值函数，并返回结果。
- 测试。
- 考虑对第一步得到的访问函数使用函数改名（124），以便更好反映其用途。

- 考虑应用将引用对象改为值对象（252）或将值对象改为引用对象（256），明确指出新对象的角色是值对象还是引用对象。

范例

我将从一个简单的订单（`Order`）类开始。该类从一个简单的记录结构里读取所需的数据，这其中有一个订单优先级（`priority`）字段，它是以字符串的形式被读入的。

class Order...

```
constructor(data) {
  this.priority = data.priority;
  // more initialization
```

客户端代码有些地方是这么用它的：

客户端...

```
highPriorityCount = orders.filter(o => "high" === o.priority
                                   || "rush" === o.priority)
                          .length;
```

无论何时，当我与一个数据值打交道时，第一件事一定是对它使用封装变量（132）。

class Order...

```
get priority()        {return this._priority;}
set priority(aString) {this._priority = aString;}
```

现在构造函数中第一行初始化代码就会使用我刚刚创建的设值函数了。

这使它成了一个自封装的字段，因此我暂可放任原来的引用点不理，先对字段进行处理。

接下来我为优先级字段创建一个简单的值类（value class）。该类应该有一个构造函数接收值字段，并提供一个返回字符串的转换函数。

```
class Priority {
  constructor(value) {this._value = value;}
  toString() {return this._value;}
}
```

这里的转换函数我更倾向于使用`toString`而不用取值函数（`value`）。对类的客户端而言，一个返回字符串描述的API应该更能传达“发生了数据转

换”的信息，而使用取值函数取用一个字段就缺乏这方面的感觉。

然后我要修改访问函数，使其用上新创建的类。

class Order...

```
get priority()        {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString)
```

提炼出Priority类后，我发觉现在Order类上的取值函数命名有点儿误导人了。它确实还是返回了优先级信息，但却是一个字符串描述，而不是一个Priority对象。于是我立即对它应用了函数改名（124）。

class Order...

```
get priorityString() {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString)
```

客户端...

```
highPriorityCount = orders.filter(o => "high" === o.priorityS
                                    || "rush" === o.prioritySt
                          .length;
```

这里设值函数的名字倒没有使我不满，因为函数的参数能够清晰地表达其意图。

到此为止，正式的重构手法就结束了。不过当我进一步查看优先级字段的客户端时，我在想让它们直接使用Priority对象是否会更好。于是，我着手在订单类上添加一个取值函数，让它直接返回新建的Priority对象。

class Order...

```
get priority()        {return this._priority;}
get priorityString()  {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString)
```

客户端...

```
highPriorityCount = orders.filter(o => "high" === o.priority.
                                    || "rush" === o.priority.t
                          .length;
```

随着Priority对象在别处也有了用处，我开始支持让Order类的客户端拿着Priority实例来调用设值函数，这可以通过调整Priority类的构造函数实现。

class Priority...

```
constructor(value) {
  if (value instanceof Priority) return value;
  this._value = value;
}
```

这样做的意义在于，现在新的Priority类可以容纳更多业务行为——无论是新的业务代码，还是从别处搬移过来的。这里有些例子，它会校验优先级的传入值，支持一些比较逻辑。

class Priority...

```
constructor(value) {
  if (value instanceof Priority) return value;
  if (Priority.legalValues().includes(value))
    this._value = value;
  else
    throw new Error(`<${value}> is invalid for Priority`);
}
toString() {return this._value;}
get _index() {return Priority.legalValues().findIndex(s => s
static legalValues() {return ['low', 'normal', 'high', 'rush'
```

```
equals(other) {return this._index === other._index;}
higherThan(other) {return this._index > other._index;}
lowerThan(other) {return this._index < other._index;}
```

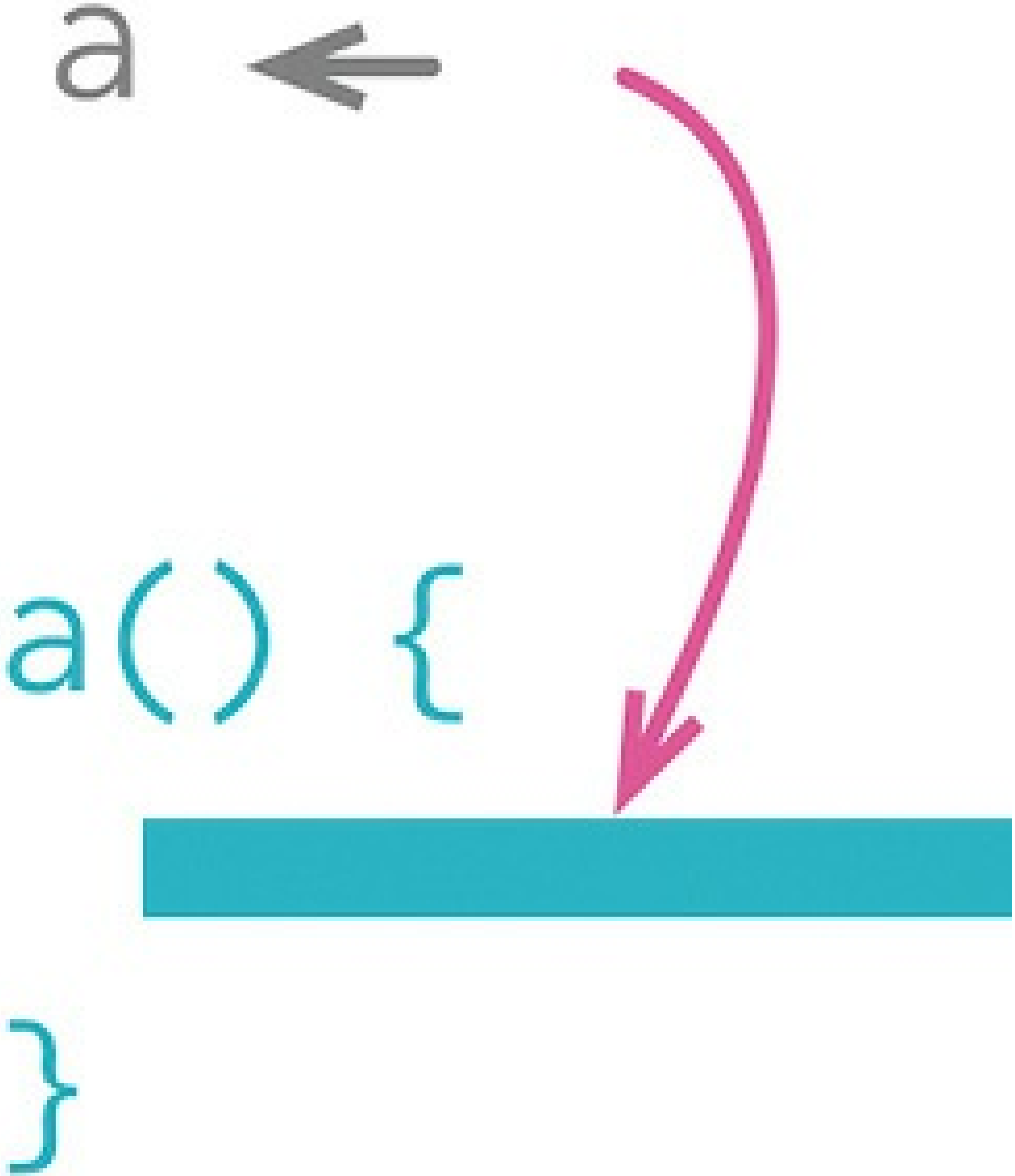
修改的过程中，我发觉它实际上已经担负起值对象（value object）的角色，因此我又为它添加了一个`equals`方法，并确保它的值不可修改。

加上这些行为后，我可以让客户端代码读起来含义更清晰。

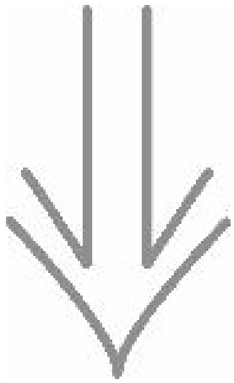
客户端...

```
highPriorityCount = orders.filter(o => o.priority.higherThan(
                          .length;
```

7.4　以查询取代临时变量（Replace Temp with Query）


a
a() {
}

```
const basePrice = this._quantity * this._itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```



```
get basePrice() {this._quantity * this._itemPrice;}

...

if (this.basePrice > 1000)
  return this.basePrice * 0.95;
else
  return this.basePrice * 0.98;
```

动机

临时变量的一个作用是保存某段代码的返回值，以便在函数的后面部分使用它。临时变量允许我引用之前的值，既能解释它的含义，还能避免对代码进行重复计算。但尽管使用变量很方便，很多

时候还是值得更进一步，将它们抽取成函数。

如果我正在分解一个冗长的函数，那么将变量抽取到函数里能使函数的分解过程更简单，因为我就不再需要将变量作为参数传递给提炼出来的小函数。将变量的计算逻辑放到函数中，也有助于在提炼得到的函数与原函数之间设立清晰的边界，这能帮我发现并避免难缠的依赖及副作用。

改用函数还让我避免了在多个函数中重复编写计算逻辑。每当我在不同的地方看见同一段变量的计算逻辑，我就会想方设法将它们挪到同一个函数里。

这项重构手法在类中施展效果最好，因为类为待提炼函数提供了一个共同的上下文。如果不是在类中，我很可能会在顶层函数中拥有过多参数，这将冲淡提炼函数所能带来的诸多好处。使用嵌套的小函数可以避免这个问题，但又限制了我在相关函数间分享逻辑的能力。

以查询取代临时变量（178）手法只适用于处理某些类型的临时变量：那些只被计算一次且之后不再被修改的变量。最简单的情况是，这个临时变量只被赋值一次，但在更复杂的代码片段里，变量也可能被多次赋值——此时应该将这些计算代码一并提炼到查询函数中。并且，待提炼的逻辑多次计

算同样的变量时，应该能得到相同的结果。因此，对于那些做快照用途的临时变量（从变量名往往可见端倪，比如oldAddress这样的名字），就不能使用本手法。

做法

- 检查变量在使用前是否已经完全计算完毕，检查计算它的那段代码是否每次都能得到一样的值。
- 如果变量目前不是只读的，但是可以改造成只读变量，那就先改造它。
- 测试。
- 将为变量赋值的代码段提炼成函数。

如果变量和函数不能使用同样的名字，那么先为函数取个临时的名字。

确保待提炼函数没有副作用。若有，先应用将查询函数和修改函数分离（306）手法隔离副作用。

- 测试。
- 应用内联变量（123）手法移除临时变量。

范例

这里有一个简单的订单类。

class Order...

```
  constructor(quantity, item) {
    this._quantity = quantity;
    this._item = item;
  }

  get price() {
    var basePrice = this._quantity * this._item.price;
    var discountFactor = 0.98;
    if (basePrice > 1000) discountFactor -= 0.03;
    return basePrice * discountFactor;
  }
}
```

我希望把basePrice和discountFactor两个临时变量变成函数。

先从basePrice开始，我先把它声明成const并运行测试。这可以很好地防止我遗漏了对变量的其他赋值点——对于这么个小函数是不太可能的，但

当我处理更大的函数时就不一定了。

class Order...

```
  constructor(quantity, item) {
    this._quantity = quantity;
    this._item = item;
  }

  get price() {
    const basePrice = this._quantity * this._item.price;
    var discountFactor = 0.98;
    if (basePrice > 1000) discountFactor -= 0.03;
    return basePrice * discountFactor;
  }
}
```

然后我把赋值操作的右边提炼成一个取值函数。

class Order...

```
get price() {
  const basePrice = this.basePrice;
  var discountFactor = 0.98;
  if (basePrice > 1000) discountFactor -= 0.03;
  return basePrice * discountFactor;
}

  get basePrice() {
    return this._quantity * this._item.price;
  }
```

测试，然后应用内联变量（123）。

class Order...

```
get price() {
  ~~const basePrice = this.basePrice;~~
  var discountFactor = 0.98;
  if (this.basePrice > 1000) discountFactor -= 0.03;
  return this.basePrice * discountFactor;
}
```

接下来我对discountFactor重复同样的步骤，先是应用提炼函数（106）。

class Order...

```
get price() {
  const discountFactor = this.discountFactor;
  return this.basePrice * discountFactor;
}

  get discountFactor() {
    var discountFactor = 0.98;
    if (this.basePrice > 1000) discountFactor -= 0.03;
    return discountFactor;
}
```

这里我需要将对discountFactor的两处赋值一起搬移到新提炼的函数中，之后就可以将原变量一

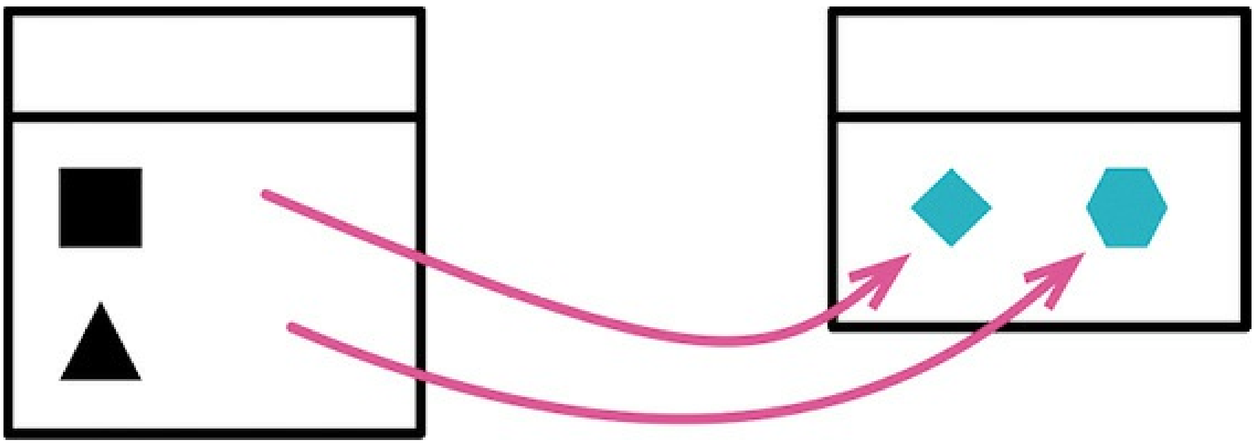
起声明为`const`。

然后，内联变量：

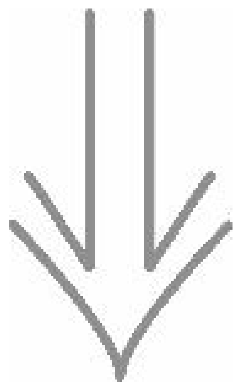
```
get price() {
  return this.basePrice * this.discountFactor;
}
```

7.5　提炼类（Extract Class）

反向重构：内联类（186）



```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber()   {return this._officeNumber;}
```



```
class Person {
```

```
get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber()   {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number()   {return this._number;}
}
```

动机

你也许听过类似这样的建议：一个类应该是一个清晰的抽象，只处理一些明确的责任，等等。但是在实际工作中，类会不断成长扩展。你会在这儿加入一些功能，在那儿加入一些数据。给某个类添加一项新责任时，你会觉得不值得为这项责任分离出一个独立的类。于是，随着责任不断增加，这个类会变得过分复杂。很快，你的类就会变成一团乱麻。

设想你有一个维护大量函数和数据的类。这样的类往往因为太大而不易理解。此时你需要考虑哪些部分可以分离出去，并将它们分离到一个独立的类中。如果某些数据和某些函数总是一起出现，某些数据经常同时变化甚至彼此相依，这就表示你应该将它们分离出去。一个有用的测试就是问你自己，如果你搬移了某些字段和函数，会发生什么

事？其他字段和函数是否因此变得无意义？

另一个往往在开发后期出现的信号是类的子类化方式。如果你发现子类化只影响类的部分特性，或如果你发现某些特性需要以一种方式来子类化，某些特性则需要以另一种方式子类化，这就意味着你需要分解原来的类。

做法

- 决定如何分解类所负的责任。
- 创建一个新的类，用以表现从旧类中分离出来的责任。

  如果旧类剩下的责任与旧类的名称不符，为旧类改名。

- 构造旧类时创建一个新类的实例，建立“从旧类访问新类”的连接关系。
- 对于你想搬移的每一个字段，运用搬移字段（207）搬移之。每次更改后运行测试。

- 使用搬移函数（198）将必要函数搬移到新类。先搬移较低层函数（也就是“被其他函数调用”多于“调用其他函数”者）。每次更改后运行测试。
- 检查两个类的接口，去掉不再需要的函数，必要时为函数重新取一个适合新环境的名字。
- 决定是否公开新的类。如果确实需要，考虑对新类应用将引用对象改为值对象（252）使其成为一个值对象。

范例

我们从一个简单的Person类开始。

class Person...

```
get name()    {return this._name;}
set name(arg) {this._name = arg;}
get telephoneNumber() {return `(${this.officeAreaCode}) ${thi
get officeAreaCode()    {return this._officeAreaCode;}
set officeAreaCode(arg)  {this._officeAreaCode = arg;}
get officeNumber() {return this._officeNumber;}
set officeNumber(arg) {this._officeNumber = arg;}
```

这里，我可以将与电话号码相关的行为分离到

一个独立的类中。首先，我要定义一个空的TelephoneNumber类来表示“电话号码”这个概念：

```
class TelephoneNumber {
}
```

易如反掌！接着，我要在构造Person类时创建TelephoneNumber类的一个实例。

class Person...

```
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}
```

class TelephoneNumber...

```
get officeAreaCode()    {return this._officeAreaCode;}
set officeAreaCode(arg) {this._officeAreaCode = arg;}
```

现在，我运用搬移字段（207）搬移一个字段。

class Person...

```
get officeAreaCode()    {return this._telephoneNumber.officeA
set officeAreaCode(arg) {this._telephoneNumber.officeAreaCode
```

再次运行测试，然后我对下一个字段进行同样处理。

class TelephoneNumber...

```
get officeNumber() {return this._officeNumber;}
set officeNumber(arg) {this._officeNumber = arg;}
```

class Person...

```
get officeNumber() {return this._telephoneNumber.officeNumber
set officeNumber(arg) {this._telephoneNumber.officeNumber = a
```

再次测试，然后再搬移对电话号码的取值函数。

class TelephoneNumber...

```
get telephoneNumber() {return `(${this.officeAreaCode}) ${thi
```

class Person...

```
get telephoneNumber() {return this._telephoneNumber.telephone
```

现在我需要做些清理工作。“电话号码”显然不该拥有“办公室”（office）的概念，因此我得重命名一下变量。

class TelephoneNumber...

```
get areaCode()    {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}

get number()    {return this._number;}
set number(arg) {this._number = arg;}
```

class Person...

```
get officeAreaCode()    {return this._telephoneNumber.areaCod
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

TelephoneNumber类上有一个对自己（telephone number）的取值函数也没什么道理，因此我又对它应用函数改名（124）。

class TelephoneNumber...

```
toString() {return `(${this.areaCode}) ${this.number}`;}
```

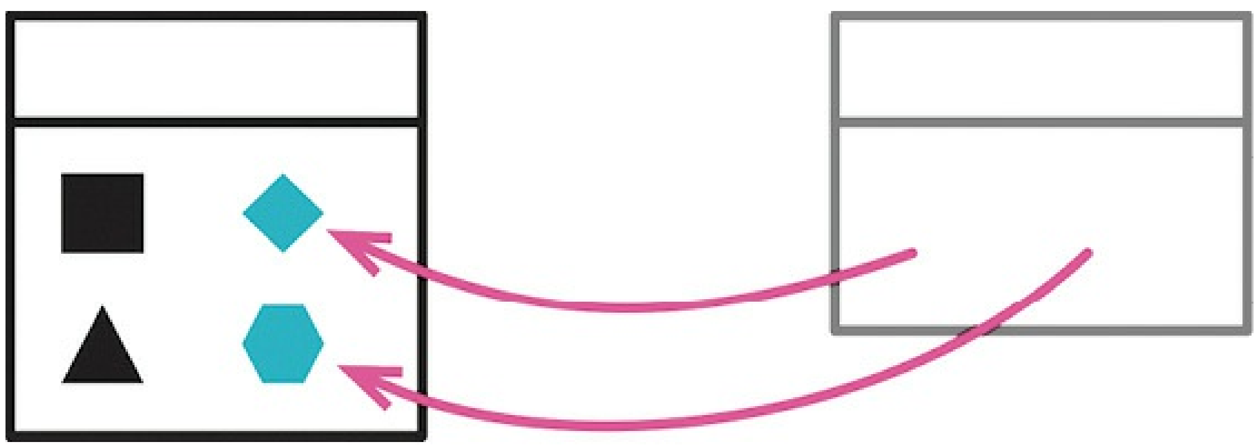
class Person...

```
get telephoneNumber() {return this._telephoneNumber.toString(
```

“电话号码”对象一般还具有复用价值，因此我考虑将新提炼的类暴露给更多的客户端。需要访问TelephoneNumber对象时，只须把Person类中那些office开头的访问函数搬移过来并略作修改即可。但这样TelephoneNumber就更像一个值对象（Value Object）[mf-vo]了，因此我会先对它使用将引用对象改为值对象（252）（那个重构手法所用的范例，正是基于本章电话号码例子的延续）。

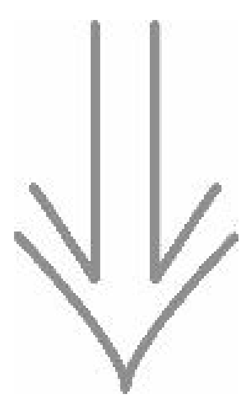
7.6 内联类（Inline Class）

反向重构：提炼类（182）



```
class Person {

get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber()   {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number() {return this._number;}
}
```



```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber()   {return this._officeNumber;}
```

动机

内联类正好与提炼类（182）相反。如果一个类不再承担足够责任，不再有单独存在的理由（这通常是因为此前的重构动作移走了这个类的责任），我就会挑选这一“萎缩类”的最频繁用户（也是一个类），以本手法将“萎缩类”塞进另一个类中。

应用这个手法的另一个场景是，我手头有两个类，想重新安排它们肩负的职责，并让它们产生关联。这时我发现先用本手法将它们内联成一个类再用提炼类（182）去分离其职责会更加简单。这是重新组织代码时常用的做法：有时把相关元素一口气搬移到位更简单，但有时先用内联手法合并各自的上下文，再使用提炼手法再次分离它们会更合适。

做法

- 对于待内联类（源类）中的所有public函数，在目标类上创建一个对应的函数，新创建的所有函数应该直接委托至源类。
- 修改源类public方法的所有引用点，令它们调用目标类对应的委托方法。每次更改后运行测试。
- 将源类中的函数与数据全部搬移到目标类，每次修改之后进行测试，直到源类变成空壳为止。
- 删除源类，为它举行一个简单的“丧礼”

范例

下面这个类存储了一次物流运输（shipment）的若干跟踪信息（tracking information）。

```
class TrackingInformation {
  get shippingCompany()    {return this._shippingCompany;}
  set shippingCompany(arg) {this._shippingCompany = arg;}
  get trackingNumber()    {return this._trackingNumber;}
  set trackingNumber(arg) {this._trackingNumber = arg;}
  get display()            {
    return `${this.shippingCompany}: ${this.trackingNumber}`;
  }
}
```

它作为Shipment（物流）类的一部分被使用。

class Shipment...

```
get trackingInfo() {
  return this._trackingInformation.display;
}
get trackingInformation() {return this._trackingInformation;}
set trackingInformation(aTrackingInformation) {
  this._trackingInformation = aTrackingInformation;
}
```

TrackingInformation类过去可能有很多光荣职责，但现在我觉得它已不再能肩负起它的责任，因此我希望将它内联到Shipment类里。

首先，我要寻找TrackingInformation类的方法有哪些调用点。

调用方...

```
aShipment.trackingInformation.shippingCompany = request.vendo
```

我将开始将源类的类似函数全都搬移到Shipment里去，但我的做法与做搬移函数（198）时略微有些不同。这里，我先在Shipment

类里创建一个委托方法，并调整客户端代码，使其调用这个委托方法。

class Shipment...

```
  set shippingCompany(arg) {this._trackingInformation.shippin
```

调用方...

```
aShipment.trackingInformation.shippingCompany = request.vendo
```

对于TrackingInformation类中所有为客户端调用的方法，我将施以相同的手法。这之后，我就可以将源类中的所有东西都搬移到Shipment类中去。

我先对display方法应用内联函数（115）手法。

class Shipment...

```
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
```

```
}
```

再继续搬移“收货公司”（shipping company）字段。

```
get shippingCompany() {return this._trackingInformation._ship
set shippingCompany(arg) {this._trackingInformation._shipping
```

我并未遵循搬移字段（207）的全部步骤，因为此处我只是改由目标类Shipment来引用shippingCompany，那些从源类搬移引用至目标类的步骤在此并不需要。

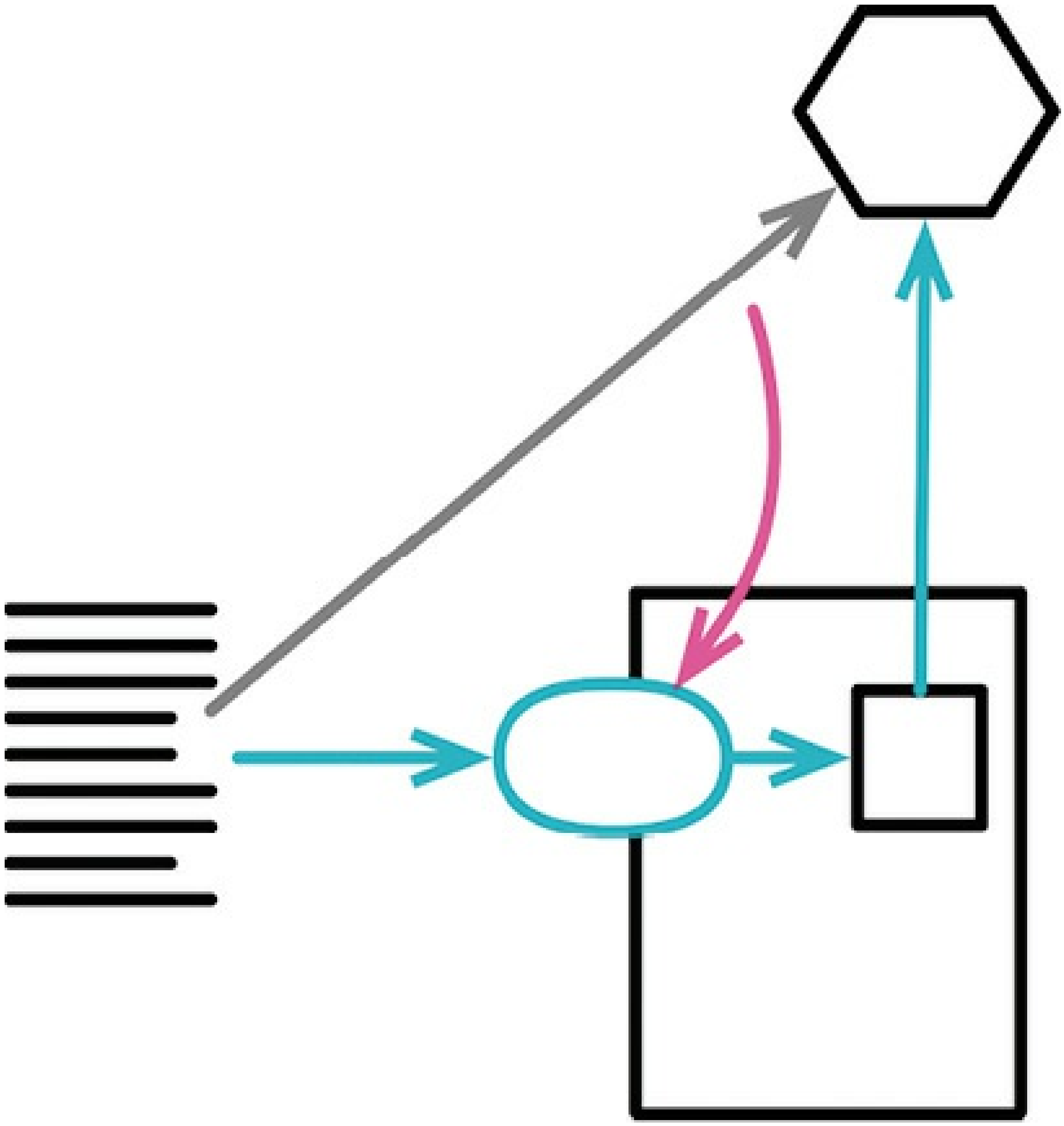
我会继续相同的手法，直到所有搬迁工作完成为止。那时，我就可以删除TrackingInformation类了。

class Shipment...

```
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
}
get shippingCompany()    {return this._shippingCompany;}
set shippingCompany(arg) {this._shippingCompany = arg;}
get trackingNumber()    {return this._trackingNumber;}
set trackingNumber(arg) {this._trackingNumber = arg;}
```

7.7 隐藏委托关系（Hide Delegate）

反向重构：移除中间人（192）



```
manager = aPerson.department.manager;
```

```
manager = aPerson.manager;

class Person {
  get manager() {return this.department.manager;}
```

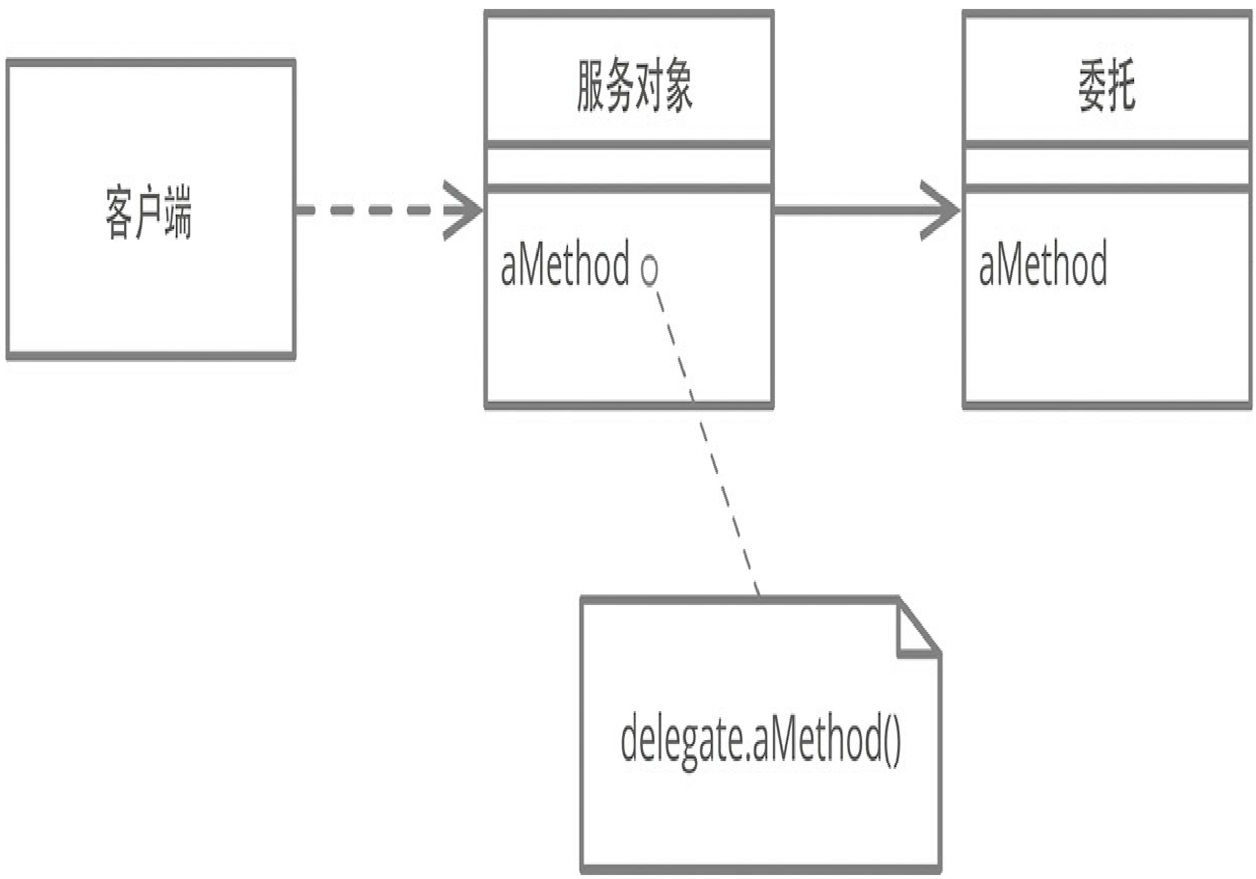
动机

一个好的模块化的设计，“封装”即使不是其最关键特征，也是最关键特征之一。“封装”意味着每个模块都应该尽可能少了解系统的其他部分。如此一来，一旦发生变化，需要了解这一变化的模块就会比较少——这会使变化比较容易进行。

当我们初学面向对象技术时就被教导，封装意味着应该隐藏自己的字段。随着经验日渐丰富，你会发现，有更多可以（而且值得）封装的东西。

如果某些客户端先通过服务对象的字段得到另

一个对象（受托类），然后调用后者的函数，那么客户就必须知晓这一层委托关系。万一受托类修改了接口，变化会波及通过服务对象使用它的所有客户端。我可以在服务对象上放置一个简单的委托函数，将委托关系隐藏起来，从而去除这种依赖。这么一来，即使将来委托关系发生变化，变化也只会影响服务对象，而不会直接波及所有客户端。



做法

- 对于每个委托关系中的函数，在服务对象端建立一个简单的委托函数。
- 调整客户端，令它只调用服务对象提供的函数。每次调整后运行测试。
- 如果将来不再有任何客户端需要取用`Delegate`（受托类），便可移除服务对象中的相关访问函数。
- 测试。

范例

本例从两个类开始，代表“人”的`Person`和代表“部门”的`Department`。

class Person...

```
constructor(name) {
  this._name = name;
}
get name() {return this._name;}
get department()    {return this._department;}
set department(arg) {this._department = arg;}
```

class Department...

```
get chargeCode() {return this._chargeCode;}
set chargeCode(arg) {this._chargeCode = arg;}
get manager() {return this._manager;}
set manager(arg) {this._manager = arg;}
```

有些客户端希望知道某人的经理是谁，为此，它必须先取得Department对象。

客户端代码...

```
manager = aPerson.department.manager;
```

这样的编码就对客户端揭露了Department的工作原理，于是客户知道：Department负责追踪“经理”这条信息。如果对客户隐藏Department，可以减少耦合。为了这一目的，我在Person中建立一个简单的委托函数。

class Person...

```
get manager() {return this._department.manager;}
```

现在，我得修改Person的所有客户端，让它们改用新函数：

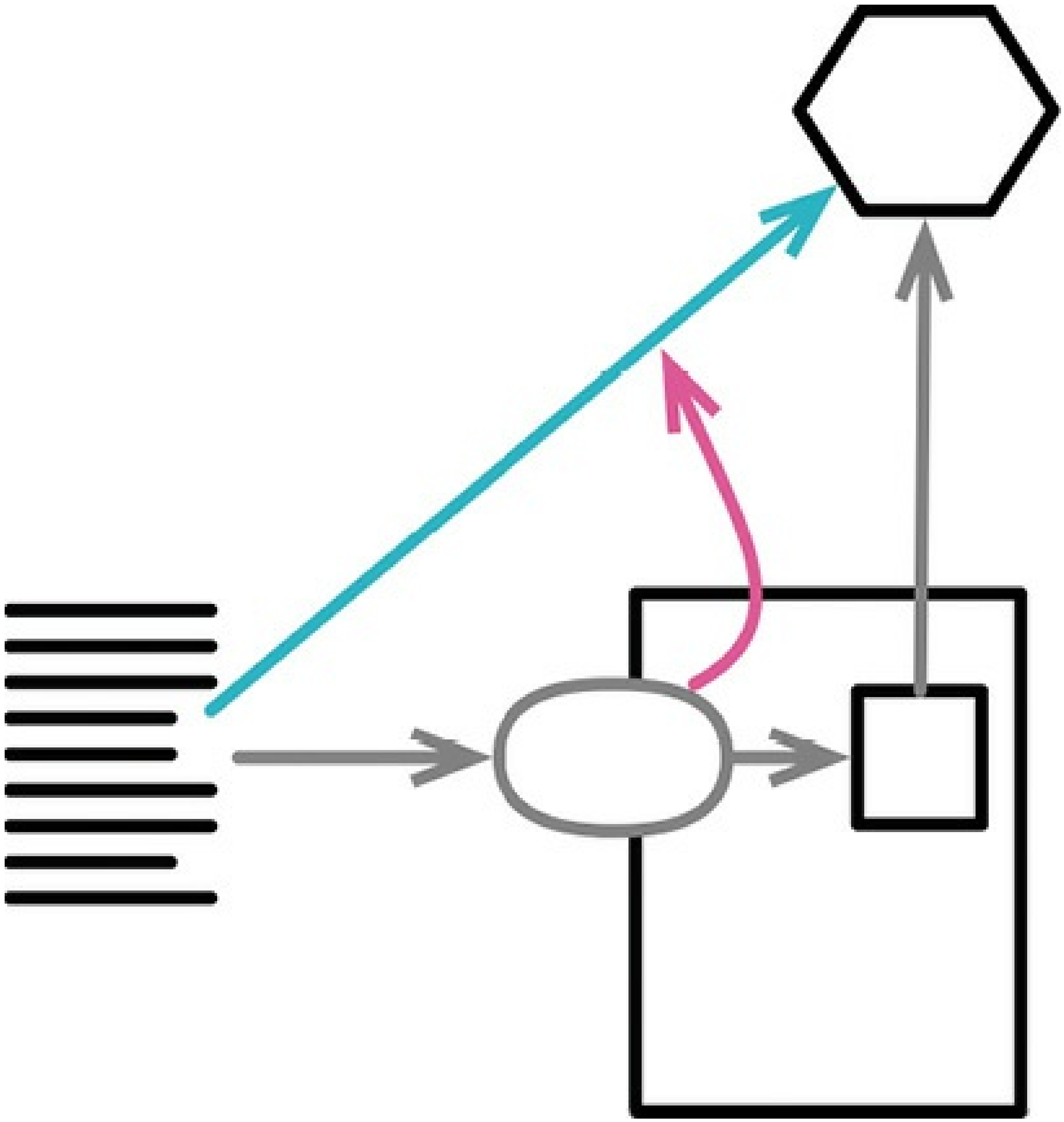
客户端代码…

`manager = aPerson`~~`.department.`~~`manager;`

只要完成了对Department所有函数的修改，并相应修改了Person的所有客户端，我就可以移除Person中的department访问函数了。

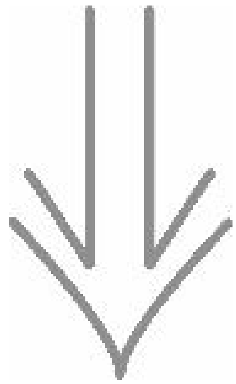
7.8　移除中间人（Remove Middle Man）

反向重构：隐藏委托关系（189）



```
manager = aPerson.manager;

class Person {
  get manager() {return this.department.manager;}
```



```
manager = aPerson.department.manager;
```

动机

在隐藏委托关系（189）的“动机”一节中，我谈到了“封装受托对象”的好处。但是这层封装也是有代价的。每当客户端要使用受托类的新特性时，你就必须在服务端添加一个简单委托函数。随着受托类的特性（功能）越来越多，更多的转发函数就会使人烦躁。服务类完全变成了一个中间人（81），此时就应该让客户直接调用受托类。（这个味道通常在人们狂热地遵循迪米特法则时悄然出现。我总觉得，如果这条法则当初叫作“偶尔有用的迪米特建议”，如今能少很多烦恼。）

很难说什么程度的隐藏才是合适的。还好，有了隐藏委托关系（189）和删除中间人，我大可不

必操心这个问题，因为我可以在系统运行过程中不断进行调整。随着代码的变化，“合适的隐藏程度”这个尺度也相应改变。6个月前恰如其分的封装，现今可能就显得笨拙。重构的意义就在于：你永远不必说对不起——只要把出问题的地方修补好就行了。

做法

- 为受托对象创建一个取值函数。
- 对于每个委托函数，让其客户端转为连续的访问函数调用。每次替换后运行测试。

替换完委托方法的所有调用点后，你就可以删掉这个委托方法了。

这能通过可自动化的重构手法来完成，你可以先对受托字段使用封装变量（132），再应用内联函数（115）内联所有使用它的函数。

范例

我又要从一个Person类开始了，这个类通过维护一个部门对象来决定某人的经理是谁。（如果你一口气读完本书的好几章，可能会发现每个“人与部门”的例子都出奇地相似。）

客户端代码...

```
manager = aPerson.manager;
```

class Person...

```
get manager() {return this._department.manager;}
```

class Department...

```
get manager() {return this._manager;}
```

像这样，使用和封装Department都很简单。但如果大量函数都这么做，我就不得不在Person之中安置大量委托行为。这就该是移除中间人的时候了。首先在Person中建立一个函数，用于获取受托

对象。

class Person...

```
get department() {return this._department;}
```

然后逐一处理每个客户端，使它们直接通过受托对象完成工作。

客户端代码**...**

```
manager = aPerson.department.manager;
```

完成对客户端引用点的替换后，我就可以从Person中移除manager方法了。我可以重复此法，移除Person中其他类似的简单委托函数。

我可以混用两种用法。有些委托关系非常常用，因此我想保住它们，这样可使客户端代码调用更友好。何时应该隐藏委托关系，何时应该移除中间人，对我而言没有绝对的标准——代码环境自然会给出该使用哪种手法的线索，具备思考能力的程序员应能分辨出何种手法更佳。

如果手边在用自动化的重构工具，那么本手法的步骤有一个实用的变招：我可以先对department应用封装变量（132）。这样可让manager的取值函数调用department的取值函数。

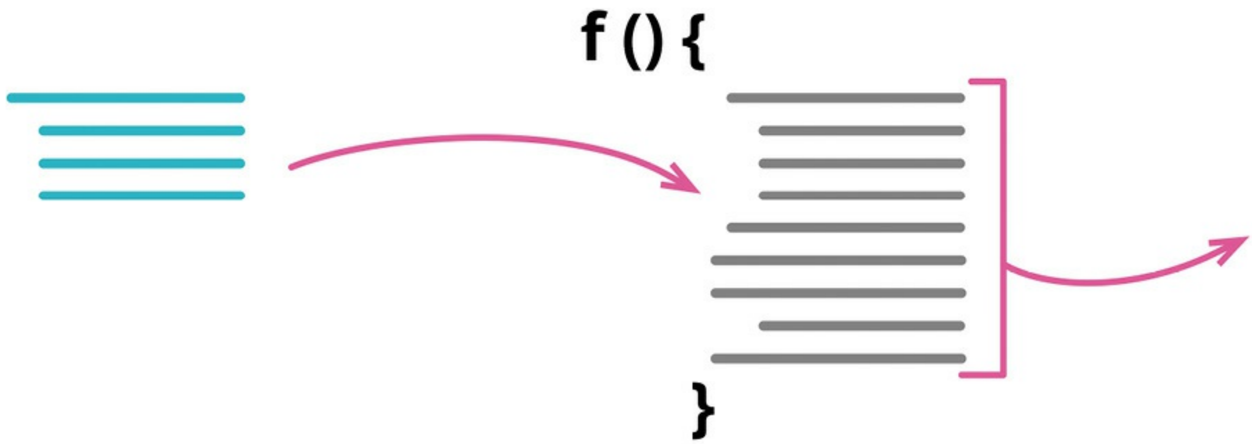
class Person...

```
get manager() {return this.department.manager;}
```

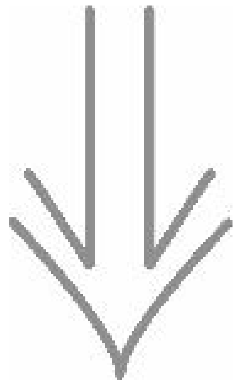
在JavaScript中，调用取值函数的语法跟取用普通字段看起来很像，但通过移除department字段的下划线，我想表达出这里是调用了取值函数而非直接取用字段的区别。

然后我对manager方法应用内联函数（115），一口气替换它的所有调用点。

7.9　替换算法（Substitute Algorithm）




```
function foundPerson(people) {
  for(let i = 0; i < people.length; i++) {
    if (people[i] === "Don") {
      return "Don";
    }
    if (people[i] === "John") {
      return "John";
    }
    if (people[i] === "Kent") {
      return "Kent";
    }
  }
  return "";
}
```



```
function foundPerson(people) {
  const candidates = ["Don", "John", "Kent"];
  return people.find(p => candidates.includes(p)) || '';
}
```

动机

我从没试过给猫剥皮，听说有好几种方法，我敢肯定，其中某些方法会比另一些简单。算法也是如此。如果我发现做一件事可以有更清晰的方式，我就会用比较清晰的方式取代复杂的方式。“重构”可以把一些复杂的东西分解为较简单的小块，但有时你就必须壮士断腕，删掉整个算法，代之以较简单的算法。随着对问题有了更多理解，我往往会发现，在原先的做法之外，有更简单的解决方案，此时我就需要改变原先的算法。如果我开始使用程序库，而其中提供的某些功能/特性与我自己的代码重复，那么我也需要改变原先的算法。

有时我会想修改原先的算法，让它去做一件与原先略有差异的事。这时候可以先把原先的算法替换为一个较易修改的算法，这样后续的修改会轻松许多。

使用这项重构手法之前，我得确定自己已经尽可能分解了原先的函数。替换一个巨大且复杂的算法是非常困难的，只有先将它分解为较简单的小型函数，我才能很有把握地进行算法替换工作。

做法

- 整理一下待替换的算法，保证它已经被抽取到一个独立的函数中。
- 先只为这个函数准备测试，以便固定它的行为。
- 准备好另一个（替换用）算法。
- 执行静态检查。
- 运行测试，比对新旧算法的运行结果。如果测试通过，那就大功告成；否则，在后续测试和调试过程中，以旧算法为比较参照标准。

第8章　搬移特性

到目前为止，我介绍的重构手法都是关于如何新建、移除或重命名程序的元素。此外，还有另一种类型的重构也很重要，那就是在不同的上下文之间搬移元素。我会通过搬移函数（198）手法在类与其他模块之间搬移函数，对于字段可用搬移字段（207）手法做类似的搬移。

有时我还需要单独对语句进行搬移，调整它们的顺序。搬移语句到函数（213）和搬移语句到调用者（217）可用于将语句搬入函数或从函数中搬出；如果需要在函数内部调整语句的顺序，那么移动语句（223）就能派上用场。有时一些语句做的事已有现成的函数代替，那时我就能以函数调用取代内联代码（222）消除重复。

对付循环，我有两个常用的手法：拆分循环（227）可以确保每个循环只做一件事，以管道取代循环（231）则可以直接消灭整个循环。

最后这项手法，我相信一定会是任何一个合格程序员的至爱，那就是移除死代码（237）。没什

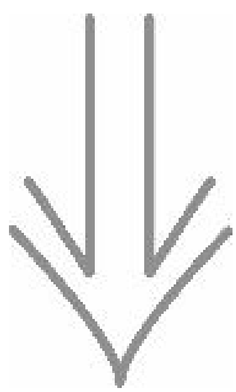
么能比手刃一段长长的无用代码更令一个程序员感到满足的了。

8.1　搬移函数（Move Function）

曾用名：搬移函数（Move Method）



```
class Account {
  get overdraftCharge() {...}
```



```
class AccountType {
    get overdraftCharge() {...}
```

动机

模块化是优秀软件设计的核心所在，好的模块化能够让我在修改程序时只需理解程序的一小部分。为了设计出高度模块化的程序，我得保证互相关联的软件要素都能集中到一块，并确保块与块之间的联系易于查找、直观易懂。同时，我对模块设计的理解并不是一成不变的，随着我对代码的理解加深，我会知道那些软件要素如何组织最为恰当。要将这种理解反映到代码上，就得不断地搬移这些元素。

任何函数都需要具备上下文环境才能存活。这个上下文可以是全局的，但它更多时候是由某种形式的模块所提供的。对一个面向对象的程序而言，类作为最主要的模块化手段，其本身就能充当函数的上下文；通过嵌套的方式，外层函数也能为内层函数提供一个上下文。不同的语言提供的模块化机制各不相同，但这些模块的共同点是，它们都能为函数提供一个赖以存活的上下文环境。

搬移函数最直接的一个动因是，它频繁引用其他上下文中的元素，而对自身上下文中的元素却关心甚少。此时，让它去与那些更亲密的元素相会，通常能取得更好的封装效果，因为系统别处就可以减少对当前模块的依赖。

同样，如果我在整理代码时，发现需要频繁调用一个别处的函数，我也会考虑搬移这个函数。有

时你在函数内部定义了一个帮助函数，而该帮助函数可能在别的地方也有用处，此时就可以将它搬移到某些更通用的地方。同理，定义在一个类上的函数，可能挪到另一个类中去更方便我们调用。

是否需要搬移函数常常不易抉择。为了做出决定，我需要仔细检查函数当前上下文与目标上下文之间的区别，需要查看函数的调用者都有谁，它自身又调用了哪些函数，被调用函数需要什么数据，等等。在搬移过程中，我通常会发现需要为一整组函数创建一个新的上下文，此时就可以用函数组合成类（144）或提炼类（182）创建一个。尽管为函数选择一个最好的去处不太容易，但决定越难做，通常说明“搬移这个函数与否”的重要性也越低。我发现好的做法是先把函数安置到某一个上下文里去，这样我就能发现它们是否契合，如果不太合适我可以再把函数搬移到别的地方。

做法

- 检查函数在当前上下文里引用的所有程序元素（包括变量和函数），考虑是否需要将它们一并搬移

如果发现有些被调用的函数也需要搬移，我通常会先搬移它们。这样可以保证移动一组函数时，总是从依赖最少的那个函数入手。

如果该函数拥有一些子函数，并且它是这些子函数的唯一调用者，那么你可以先将子函数内联进来，一并搬移到新家后再重新提炼出子函数。

- 检查待搬移函数是否具备多态性。

在面向对象的语言里，还需要考虑该函数是否覆写了超类的函数，或者为子类所覆写。

- 将函数复制一份到目标上下文中。调整函数，使它能适应新家。

如果函数里用到了源上下文（source context）中的元素，我就得将这些元素一并传递过去，要么通过函数参数，要么是将当前上下文的引用传递到新的上下文那边去。

搬移函数通常意味着，我还得给它起个新名字，使它更符合新的上下文。

- 执行静态检查。
- 设法从源上下文中正确引用目标函数。
- 修改源函数，使之成为一个纯委托函数。
- 测试。
- 考虑对源函数使用内联函数（115）

也可以不做内联，让源函数一直做委托调用。但如果调用方直接调用目标函数也不费太多周折，那么最好还是把中间人移除掉。

范例：搬移内嵌函数至顶层

让我用一个函数来举例。这个函数会计算一条GPS轨迹记录（track record）的总距离（total distance）。

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
```

```
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1],  points[i]);
    }
    return result;
  }

  function distance(p1,p2) { ... }
  function radians(degrees) { ... }
  function calculateTime() { ... }

}
```

我希望把calculateDistance函数搬移到顶层，这样我就能单独计算轨迹的距离，而不必算出汇总报告（summary）的其他部分。

我先将函数复制一份到顶层作用域中：

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result  =  0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
```

```
    }
    return result;
  }
  ...
  function distance(p1,p2) { ... }
  function radians(degrees) { ... }
  function calculateTime() { ... }

}

  function top_calculateDistance() {
    let result  =  0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1],  points[i]);
    }
    return result;
  }
```

复制函数时，我习惯为函数一并改个名，这样可让“它们有不同的作用域”这个信息显得一目了然。现在我还不想花费心思考虑它正确的名字该是什么，因此我暂且先用一个临时的名字。

此时代码依然能正常工作，但我的静态分析器要开始抱怨了，它说新函数里多了两个未定义的符号，分别是distance和points。对于points，自然是将其作为函数参数传进来。

```
function top_calculateDistance(points) {
  let result =0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1],  points[i]);
  }
  return result;
}
```

至于distance，虽然我也可以将它作为参数传进来，但也许将其计算函数calculate Distance一并搬移过来会更合适。该函数的代码如下。

function trackSummary...

```
function distance(p1,p2) {
  const EARTH_RADIUS = 3959; // in miles
  const dLat = radians(p2.lat) - radians(p1.lat);
  const dLon = radians(p2.lon) - radians(p1.lon);
  const a = Math.pow(Math.sin(dLat / 2),2)
         + Math.cos(radians(p2.lat))
         * Math.cos(radians(p1.lat))
         * Math.pow(Math.sin(dLon / 2), 2);
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
  return EARTH_RADIUS * c;
}
function radians(degrees) {
  return degrees * Math.PI / 180;
}
```

我留意到distance函数中只调用了radians函数，后者已经没有再引用当前上下文里的任何元素。因此与其将radians作为参数，我更倾向于将它也一并搬移。不过我不需要一步到位，我们可以先将这两个函数从当前上下文中搬移进calculateDistance函数里：

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
```

```
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;

   function distance(p1,p2) { ... }
   function radians(degrees) { ... }

}
```

这样做的好处是，我可以充分发挥静态检查和测试的作用，让它们帮我检查有无遗漏的东西。在这个实例中一切顺利，因此，我可以放心地将这两个函数直接复制到`top_calculateDistance`中：

```
function top_calculateDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1],  points[i]);
  }
  return result;

 function distance(p1,p2) { ... }
 function radians(degrees) { ... }

}
```

这次复制操作同样不会改变程序现有行为，但给了静态分析器更多介入的机会，增加了暴露错误

的概率。假如我在上一步没有发现distance函数内部还调用了radians函数，那么这一步就会被分析器检查出来。

现在万事俱备，是时候端出主菜了——我要在原calculateDistance函数体内调用top_calculateDistance函数：

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    return top_calculateDistance(points);
  }
```

接下来最重要的事是要运行一遍测试，看看功能是否仍然完整，函数在其新家待得是否舒适。

测试通过后，便算完成了主要任务，就好比搬家，现在大箱小箱已经全搬到新家，接下来就是将它们拆箱复位了。第一件事是决定还要不要保留原来那个只起委托作用的函数。在这个例子中，原函数的调用点不多，作为嵌套函数它们的作用范围通常也很小，因此我觉得这里大可直接移除原函数。

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = top_calculateDistance(points);
  const pace = totalTime / 60 / totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };
}
```

同时，也该是时候为这个函数认真想个名字了。因为顶层函数拥有最高的可见性，因此取个好名非常重要。totalDistance听起来不错，但还不能马上用这个名字，因为trackSummary函数中有一个同名的变量——我不觉得这个变量有保留的价值，因此我们先用内联变量（123）处理它，之后再使用改变函数声明（124）：

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const pace = totalTime / 60 / totalDistance(points) ;
  return {
    time: totalTime,
    distance: totalDistance(points),
    pace: pace
  };
}
function totalDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
}
```

如果出于某些原因，实在需要保留该变量，那么我建议将该变量改个其他的名字，比如totalDistanceCache或distance等。

由于distance函数和radians函数并未使用totalDistance中的任何变量或函数，因此我倾向于把它们也提升到顶层，也就是4个方法都放置在顶层作用域上。

```
function trackSummary(points) { ... }
function totalDistance(points) { ... }
function distance(p1,p2) { ... }
function radians(degrees) { ... }
```

有些人则更倾向于将distance和radians函数保留在totalDistance内，以便限制它们的可见性。在某些语言里，这个顾虑也许有其道理，但新的ES 2015规范为JavaScript提供了一个美妙的模块化机制，利用它来控制函数的可见性是再好不过了。通常来说，我对嵌套函数还是心存警惕的，因为很容易在里面编写一些私有数据，并且在函数之间共享，这可能会增加代码的阅读和重构难度。

范例：在类之间搬移函数

在类之间搬移函数也是一种常见场景，下面我将用一个表示“账户”的Account类来讲解。

class Account...

```
get bankCharge() {
  let result = 4.5;

if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  if (this.type.isPremium) {
    const baseCharge = 10;
    if (this.daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (this.daysOverdrawn - 7) * 0.85;
  }
  else
    return this.daysOverdrawn * 1.75;
}
```

上面的代码会根据账户类型（account type）的不同，决定不同的“透支金额计费”算法。因此，很自然会想到将overdraftCharge函数搬移到AccountType类去。

第一步要做的是：观察被overdraftCharge使用的每一项特性，考虑是否值得将它们与overdraftCharge函数一起移动。此例中我需要

让daysOverdrawn字段留在Account类中，因为它会随不同种类的账户而变化。

然后，我将overdraftCharge函数主体复制到AccountType类中，并做相应调整。

class AccountType...

```
overdraftCharge(daysOverdrawn) {
  if (this.isPremium) {
    const baseCharge  =  10;
    if (daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (daysOverdrawn - 7) * 0.85;
  }
  else
    return daysOverdrawn * 1.75;
}
```

为了使函数适应这个新家，我必须决定如何处理两个作用范围发生改变的变量。isPremium如今只需要简单地从this上获取，但daysOverdrawn怎么办呢？我是直接传值，还是把整个account对象传过来？为了方便，我选择先简单传一个值，不过如果后续还需要账户（account）对象上除了daysOverdrawn以外的更多数据，例如需要根据账户类型（account type）来决定如何从账户（account）对象上取用数据时，那么我很可能会

改变主意，转而选择传入整个account对象。

完成函数复制后，我会将原来的方法代之以一个委托调用。

class Account...

```
get bankCharge() {
  let result = 4.5;

if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  return this.type.overdraftCharge(this.daysOverdrawn);
}
```

然后下一件需要决定的事情是，是保留overdraftCharge这个委托函数，还是直接内联它？内联的话，代码会变成下面这样。

class Account...

```
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0)
    result += this.type.overdraftCharge(this.daysOverdrawn);
  return result;
}
```

在早先的步骤中，我将daysOverdrawn作为参数直接传递给overdraftCharge函数，但如若账户（account）对象上有很多数据需要传递，那我就比较倾向于直接将整个对象作为参数传递过去：

class Account...

```
get bankCharge() {
  let result = 4.5;

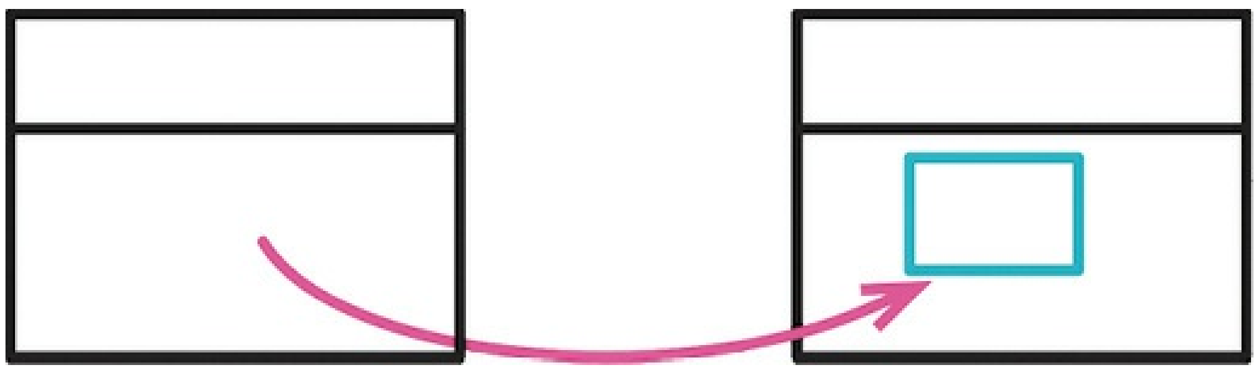
if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  return this.type.overdraftCharge(this);
}
```

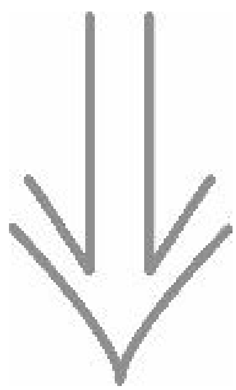
class AccountType…

```
overdraftCharge(account) {
  if (this.isPremium) {
    const baseCharge = 10;
    if (account.daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (account.daysOverdrawn - 7) * 0.85;
  }
  else
    return account.daysOverdrawn * 1.75;
}
```

8.2　搬移字段（Move Field）



```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this._discountRate;}
```



```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this.plan.discountRate;}
```

动机

编程活动中你需要编写许多代码，为系统实现特定的行为，但往往数据结构才是一个健壮程序的根基。一个适应于问题域的良好数据结构，可以让行为代码变得简单明了，而一个糟糕的数据结构则将招致许多无用代码，这些代码更多是在差劲的数据结构中间纠缠不清，而非为系统实现有用的行为。代码凌乱，势必难以理解；不仅如此，坏的数据结构本身也会掩藏程序的真实意图。

因此，好的数据结构至关重要——不过这也与编程活动的许多方面一样，它们都很难一次做对。我通常都会做些预先的设计，设法得到最恰当的数据结构，此时如果你具备一些领域驱动设计（domain-driven design）方面的经验和知识，往往有助于你更好地设计数据结构。但即便经验再丰富，技能再熟练，我仍然发现我在进行初版设计时往往还是会犯错。在不断编程的过程中，我对问题域的理解会加深，对“什么是理想的数据结构”会有更多想法。这个星期看来合理而正确的设计决策，到了下个星期可能就不再正确了。

如果我发现数据结构已经不适应于需求，就应该马上修缮它。如果容许瑕疵存在并进一步累积，

它们就会经常使我困惑，并且使代码愈来愈复杂。

我开始寻思搬移数据，可能是因为我发现每当调用某个函数时，除了传入一个记录参数，还总是需要同时传入另一条记录的某个字段一起作为参数。总是一同出现、一同作为函数参数传递的数据，最好是规整到同一条记录中，以体现它们之间的联系。修改的难度也是引起我注意的一个原因，如果修改一条记录时，总是需要同时改动另一条记录，那么说明很可能有字段放错了位置。此外，如果我更新一个字段时，需要同时在多个结构中做出修改，那也是一个征兆，表明该字段需要被搬移到一个集中的地点，这样每次只需修改一处地方。

搬移字段的操作通常是在其他更大的改动背景下发生的。实施字段搬移后，我可能会发现字段的诸多使用者应该通过目标对象来访问它，而不应该再通过源对象来访问。诸如此类的清理，我会在此后的重构中一并完成。同样，我也可能因为字段当前的一些用法而无法直接搬移它。我得先对其使用方式做一些重构，然后才能继续搬移工作。

到目前为止，我用以指称数据结构的术语都是“记录”（record），但以上论述对类和对象同样适用。类只是一种多了实例函数的记录，它与其他任何数据结构一样，都需要保持健康。不过类的实例函数确实简化了搬移数据的操作，因为它已经将

数据的存取封装到访问函数中。当我搬移数据时，只需要相应修改访问函数的引用，该字段的所有客户端依然可以正常工作。因此，如果你的数据已经用类进行了封装，那么这个重构手法会更容易进行，我下面的展开也做了“通过类封装的数据更容易搬移”这个假设。如果你要搬移的数据是裸记录，没有任何封装，虽然类似的搬移仍然能够进行，但情况就会复杂一些。

做法

- 确保源字段已经得到了良好封装。
- 测试。
- 在目标对象上创建一个字段（及对应的访问函数）。
- 执行静态检查。
- 确保源对象里能够正常引用目标对象。

也许你已经有现成的字段或方法得到目标对象。如果没有，看看是否能简单地创建一个方法完成此事。如果还是不行，你可能就得在源对象里创建一个字段，用于存储目标对象

了。这次修改可能留存很久，但你也可以只做临时修改，等到系统其他部分的重构完成就回来移除它。

- 调整源对象的访问函数，令其使用目标对象的字段。

如果源类的所有实例对象都共享对目标对象的访问权，那么可以考虑先更新源类的设值函数，让它修改源字段时，对目标对象上的字段做同样的修改。然后，再通过引入断言（302），当检测到源字段与目标字段不一致时抛出错误。一旦你确定改动没有引入任何可观察的行为变化，就可以放心地让访问函数直接使用目标对象的字段了。

- 测试。
- 移除源对象上的字段。
- 测试。

范例

我将用下面这个例子来介绍这项手法，其中Customer类代表了一位“顾客”，CustomerContract代表与顾客关联的一个“合同”。

class Customer...

```
constructor(name, discountRate) {
  this._name = name;
  this._discountRate = discountRate;
  this._contract = new CustomerContract(dateToday());
}
get discountRate() {return this._discountRate;}
becomePreferred() {
  this._discountRate += 0.03;
  // other nice things
}
applyDiscount(amount) {

return amount.subtract(amount.multiply(this._discountRate));
}
```

class CustomerContract...

```
constructor(startDate) {
  this._startDate = startDate;
}
```

我想要将折扣率（discountRate）字段从Customer类中搬移到CustomerContract里中。

第一件事情是先用封装变量（132）将对_discountRate字段的访问封装起来。

class Customer...

```
constructor(name, discountRate) {
  this._name = name;
  this._setDiscountRate(discountRate);
  this._contract = new CustomerContract(dateToday());
}
get discountRate() {return this._discountRate;}
_setDiscountRate(aNumber) {this._discountRate = aNumber;}
becomePreferred() {
  this._setDiscountRate(this.discountRate + 0.03);
  // other nice things
}
applyDiscount(amount)  {
  return amount.subtract(amount.multiply(this.discountRate));
}
```

我通过定制的applyDiscount方法来更新字段，而不是使用通常的设值函数，这是因为我不想为字段暴露一个public的设值函数。

接着我在CustomerContract中添加一个对应的字段和访问函数。

class CustomerContract...

```
constructor(startDate, discountRate) {
```

```
    this._startDate = startDate;
    this._discountRate = discountRate;
  }
  get discountRate()    {return this._discountRate;}
  set discountRate(arg) {this._discountRate = arg;}
```

接下来，我可以修改customer对象的访问函数，让它引用CustomerContract这个新添的字段。不过当我这么干时，我收到了一个错误：“Cannot set property 'discountRate' of undefined”。这是因为我们先调用了_setDiscountRate函数，而此时CustomerContract对象尚未创建出来。为了修复这个错误，我得先撤销刚刚的代码，回到上一个可工作的状态，然后再应用移动语句（223）手法，将_setDiscountRate函数调用语句挪动到创建对象的语句之后。

class Customer...

```
  constructor(name, discountRate) {
    this._name = name;
    this._setDiscountRate(discountRate);
    this._contract = new CustomerContract(dateToday());
  }
```

搬移完语句后运行一下测试。测试通过后，再次修改Customer的访问函数，让它使用_contract

对象上的discountRate字段。

class Customer...

```
get discountRate() {return this._contract.discountRate;}
_setDiscountRate(aNumber) {this._contract.discountRate = aNum
```

在JavaScript中，使用类的字段无须事先声明，因此替换完访问函数，实际上已经没有其他字段再需要我删除。

搬移裸记录

搬移字段这项重构手法对于类的实例对象通常较易进行，因为将数据访问包装到方法中，是类所天然支持的一种封装手段。如果我要搬移的字段是裸记录，并且被许多函数直接访问，那么这项重构仍然很有意义，只不过情况会复杂不少。

我可以先为记录创建相应的访问函数，并修改所有读取和更新记录的地方，使它们引用新创建的访问函数。如果待搬移的字段是不可变（immutable）的，那么我可以在设值函数中同时更

新源字段和目标字段，然后再逐步迁移读取记录的调用点。不过，我依然会尽可能先用封装记录（162）手法将记录封装成类，如此一来后续修改会更加简单。

范例：搬移字段到共享对象

现在，让我们看另外一个场景。还是那个代表“账户”的Account类，类上有一个代表“利率”的字段_interestRate。

class Account...

```
constructor(number, type, interestRate) {
  this._number = number;
  this._type = type;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

class AccountType...

```
constructor(nameString) {
  this._name = nameString;
}
```

我不希望让每个账户自己维护一个利率字段，利率应该取决于账户本身的类型，因此我想将它搬移到AccountType中去。

利率字段已经通过访问函数得到了良好的封装，因此我只需要在AccountType上创建对应的字段及访问函数即可。

class AccountType...

```
constructor(nameString, interestRate) {
  this._name = nameString;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

接着我应该着手替换Account类的访问函数，但我发现直接替换可能有个潜藏的问题。在重构之前，每个账户都自己维护一份利率数据，而现在我要让所有相同类型的账户共享同一个利率值。如果当前类型相同的账户确实拥有相同的利率，那么这次重构就能成立，因为这不会引起可观测的行为变化。但只要存在一个特例，即同一类型的账户可能有不同的利率值，那么这样的修改就不叫重构了，因为它会改变系统的可观测行为。倘若账户的数据保存在数据库中，那我就应该检查一下数据库，确

保同一类型的账户都拥有与其账户类型匹配的利率值。同时，我还可以在Account类引入断言（302），确保出现异常的利率数据时能够及时发现。

class Account...

```
  constructor(number, type, interestRate) {
    this._number = number;
    this._type = type;
    assert(interestRate === this._type.interestRate);
    this._interestRate = interestRate;
  }
  get interestRate() {return this._interestRate;}
```

我会保留这条断言，让系统先运行一段时间，看看是否会在这捕获到错误。或者，除了添加断言，我还可以将错误记录到日志里。一段时间后，一旦我对代码变得更加自信，确定它确实没有引起行为变化后，我就可以让Account直接访问AccountType上的interestRate字段，并将原来的字段完全删除了。

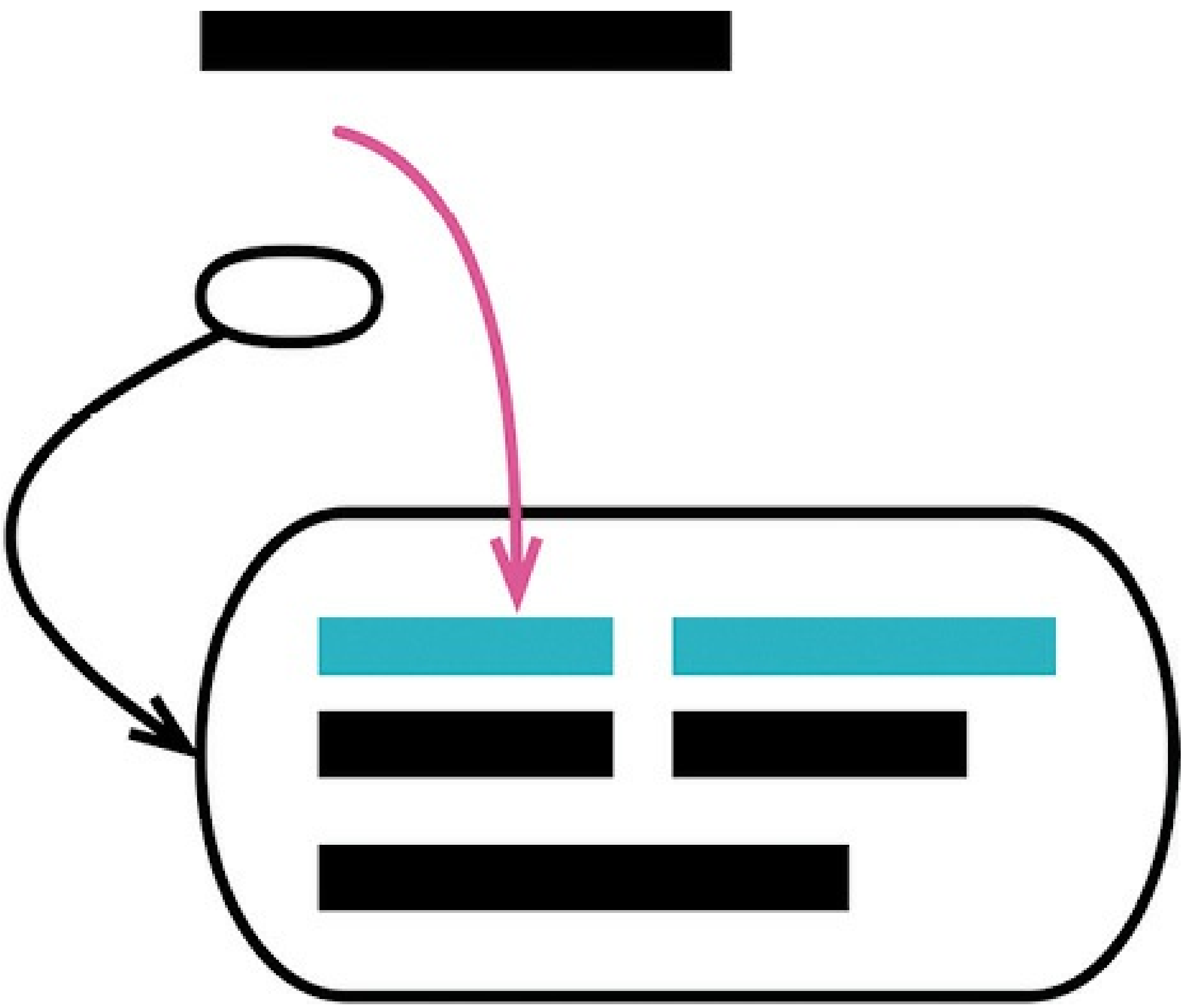
class Account...

```
  constructor(number, type) {
    this._number = number;
```

```
  this._type = type;
}
get interestRate() {return this._type.interestRate;}
```

8.3　搬移语句到函数（Move Statements into Function）

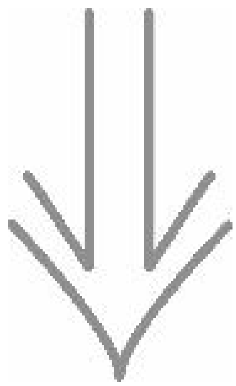
反向重构：搬移语句到调用者（217）



```
result.push(`<p>title: ${person.photo.title}</p>`);
```

```
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```



```
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

动机

要维护代码库的健康发展，需要遵守几条黄金守则，其中最重要的一条当属“消除重复”。如果我

发现调用某个函数时，总有一些相同的代码也需要每次执行，那么我会考虑将此段代码合并到函数里头。这样，日后对这段代码的修改只需改一处地方，还能对所有调用者同时生效。如果将来代码对不同的调用者需有不同的行为，那时再通过搬移语句到调用者（217）将它（或其一部分）搬移出来也十分简单。

如果某些语句与一个函数放在一起更像一个整体，并且更有助于理解，那我就会毫不犹豫地将语句搬移到函数里去。如果它们与函数不像一个整体，但仍应与函数一起执行，那我可以用提炼函数（106）将语句和函数一并提炼出去。这基本就是我下面要描述的做法了，只是下面还多了内联和改名的步骤。这些清理工作通常有其必要性，可以在完成核心步骤后再择机完成。

做法

- 如果重复的代码段离调用目标函数的地方还有些距离，则先用移动语句（223）将这些语句挪动到紧邻目标函数的位置。
- 如果目标函数仅被唯一一个源函数调用，那么只需将源函数中的重复代码段剪切并粘贴到目

标函数中即可，然后运行测试。本做法的后续步骤至此可以忽略。

- 如果函数不止一个调用点，那么先选择其中一个调用点应用提炼函数（106），将待搬移的语句与目标函数一起提炼成一个新函数。给新函数取个临时的名字，只要易于搜索即可。
- 调整函数的其他调用点，令它们调用新提炼的函数。每次调整之后运行测试。
- 完成所有引用点的替换后，应用内联函数（115）将目标函数内联到新函数里，并移除原目标函数。
- 对新函数应用函数改名（124），将其改名为原目标函数的名字。

如果你能想到更好的名字，那就用更好的那个。

范例

我将用一个例子来讲解这项手法。以下代码会生成一些关于相片（photo）的HTML：

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(`<p>title: ${person.photo.title}</p>`);
  result.push(emitPhotoData(person.photo));
  return result.join("\n");
}
function photoDiv(p) {
  return [
    "<div>",
    `<p>title:  ${p.title}</p>`,
    emitPhotoData(p),
    "</div>",
  ].join("\n");
}

function emitPhotoData(aPhoto) {
  const result = [];
  result.push(`<p>location: ${aPhoto.location}</p>`);
  result.push(`<p>date: ${aPhoto.date.toDateString()}</p>`);
  return result.join("\n");
}
```

这个例子中的emitPhotoData函数有两个调用点，每个调用点的前面都有一行类似的重复代码，用于打印与标题（title）相关的信息。我希望能消除重复，把打印标题的那行代码搬移到emitPhotoData函数里去。如果emitPhotoData只有一个调用点，那我大可直接把代码复制并粘贴过去就完事，但若调用点不止一个，那我就更倾向于用更安全的手法小步前进。

我先选择其中一个调用点，对其应用提炼函数（106）。除了我想搬移的语句，我还把emitPhotoData函数也一起提炼到新函数中。

```
function photoDiv(p) {
  return [
    "<div>",
    zznew(p),
    "</div>",
  ].join("\n");
}

function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    emitPhotoData(p),
  ].join("\n");
}
```

完成提炼后，我会逐一查看emitPhotoData的其他调用点，找到该函数与其前面的重复语句，一并换成对新函数的调用。

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(zznew(person.photo));
  return  result.join("\n");
}
```

替换完emitPhotoData函数的所有调用点后，我紧接着应用内联函数（115）将emitPhotoData函数内联到新函数中。

```
function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    `<p>location: ${p.location}</p>`,
```

```
    `<p>date: ${p.date.toDateString()}</p>`,
  ].join("\n");
}
```

最后，再对新提炼的函数应用函数改名（124），就大功告成了。

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(emitPhotoData(person.photo));
  return result.join("\n");
}

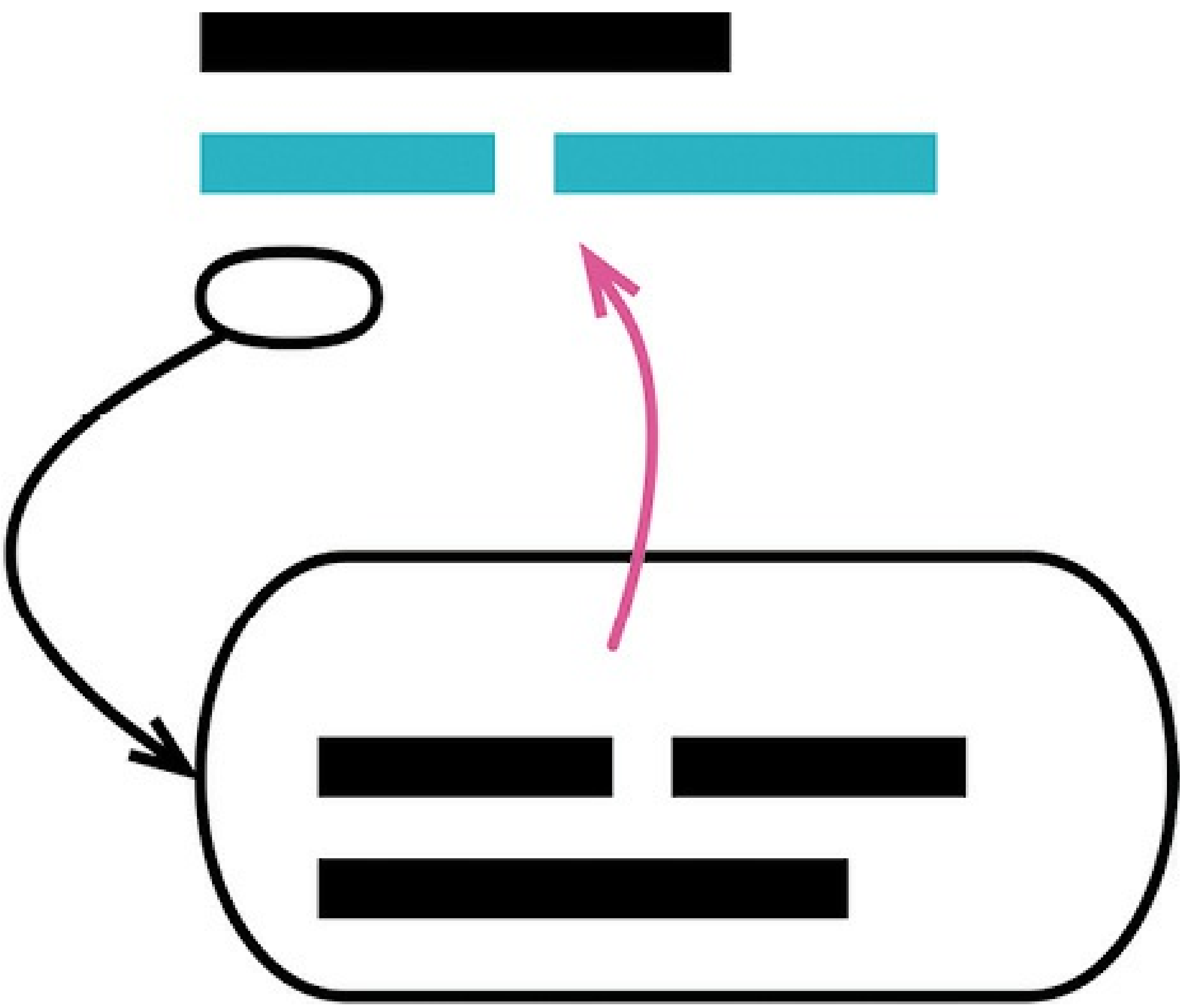
function photoDiv(aPhoto) {
  return [
    "<div>",
    emitPhotoData(aPhoto),
    "</div>",
  ].join("\n");
}

function emitPhotoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ].join("\n");
}
```

同时我会记得调整函数参数的命名，使之与我的编程风格保持一致。

8.4　搬移语句到调用者（Move Statements to Callers）

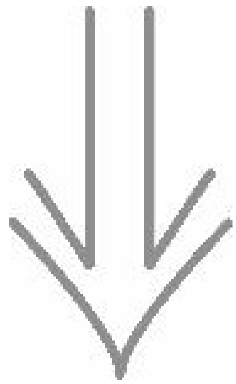
反向重构：搬移语句到函数（213）



```
emitPhotoData(outStream, person.photo);

function  emitPhotoData(outStream, photo) {
```

```
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```



```
emitPhotoData(outStream, person.photo);
outStream.write(`<p>location:       ${person.photo.location}
</p>\n`);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
}
```

动机

作为程序员，我们的职责就是设计出结构一致、抽象合宜的程序，而程序抽象能力的源泉正是来自函数。与其他抽象机制的设计一样，我们并非总能平衡好抽象的边界。随着系统能力发生演进（通常只要是有用的系统，功能都会演进），原先设定的抽象边界总会悄无声息地发生偏移。对于函

数来说，这样的边界偏移意味着曾经视为一个整体、一个单元的行为，如今可能已经分化出两个甚至是多个不同的关注点。

函数边界发生偏移的一个征兆是，以往在多个地方共用的行为，如今需要在某些调用点面前表现出不同的行为。于是，我们得把表现不同的行为从函数里挪出，并搬移到其调用处。这种情况下，我会使用移动语句（223）手法，先将表现不同的行为调整到函数的开头或结尾，再使用本手法将语句搬移到其调用点。只要差异代码被搬移到调用点，我就可以根据需要对其进行修改。

这个重构手法比较适合处理边界仅有些许偏移的场景，但有时调用点和调用者之间的边界已经相去甚远，此时便只能重新进行设计了。若果真如此，最好的办法是先用内联函数（115）合并双方的内容，调整语句的顺序，再提炼出新的函数来，以形成更合适的边界。

做法

- 最简单的情况下，原函数非常简单，其调用者也只有寥寥一两个，此时只需把要搬移的代码

从函数里剪切出来并粘贴回调用端去即可，必要的时候做些调整。运行测试。如果测试通过，那就大功告成，本手法可以到此为止。
- 若调用点不止一两个，则需要先用提炼函数（106）将你不想搬移的代码提炼成一个新函数，函数名可以临时起一个，只要后续容易搜索即可。

如果原函数是一个超类方法，并且有子类进行了覆写，那么还需要对所有子类的覆写方法进行同样的提炼操作，保证继承体系上每个类都有一份与超类相同的提炼函数。接着将子类的提炼函数删除，让它们引用超类提炼出来的函数。

- 对原函数应用内联函数（115）。
- 对提炼出来的函数应用改变函数声明（124），令其与原函数使用同一个名字。

如果你能想到更好的名字，那就用更好的那个。

范例

下面这个例子比较简单：emitPhotoData是一个函数，在两处地方被调用。

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date:      ${photo.date.toDateString()}
</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

我需要修改软件，支持listRecentPhotos函数以不同方式渲染相片的location信息，而renderPerson的行为则保持不变。为了使这次修改更容易进行，我要应用本手法，将emitPhotoData函数最后的那行代码搬移到其调用端。

一般来说，像这样简单的场景，我都会直接将emitPhotoData的最后一行剪切并粘贴到两个调

用它的函数后面。但为了演示这项重构手法如何在更复杂的场景下运作，这里我还是遵从更详细也更安全的步骤。

重构的第一步是先用提炼函数（106），将那些最终希望留在emitPhotoData函数里的语句先提炼出去。

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function  emitPhotoData(outStream, photo) {
  zztmp(outStream,  photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date:      ${photo.date.toDateString()}
</p>\n`);
}
```

新提炼出来的函数一般只会短暂存在，因此我在命名上不需要太认真，不过，取个容易搜索的名

字会很有帮助。提炼完成后运行一下测试，确保提炼出来的新函数能正常工作。

接下来，我要对emitPhotoData的调用点逐一应用内联函数（115）。先从renderPerson函数开始。

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream,  person.photo);
  outStream.write(`<p>location:     ${person.photo.location}
</p>\n`);
}
function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date:      ${photo.date.toDateString()}
</p>\n`);
}
```

然后再次运行测试，确保这次函数内联能正常工作。测试通过后，再前往下一个调用点。

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream,  person.photo);
  outStream.write(`<p>location:      ${person.photo.location}
</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      zztmp(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date:      ${photo.date.toDateString()}
</p>\n`);
}
```

至此，我就可以移除外面的emitPhotoData函数，完成内联函数（115）手法。

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream,  person.photo);
  outStream.write(`<p>location:      ${person.photo.location}
</p>\n`);
}

function listRecentPhotos(outStream, photos) {
```

```
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      zztmp(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date:      ${photo.date.toDateString()}
</p>\n`);
}
```

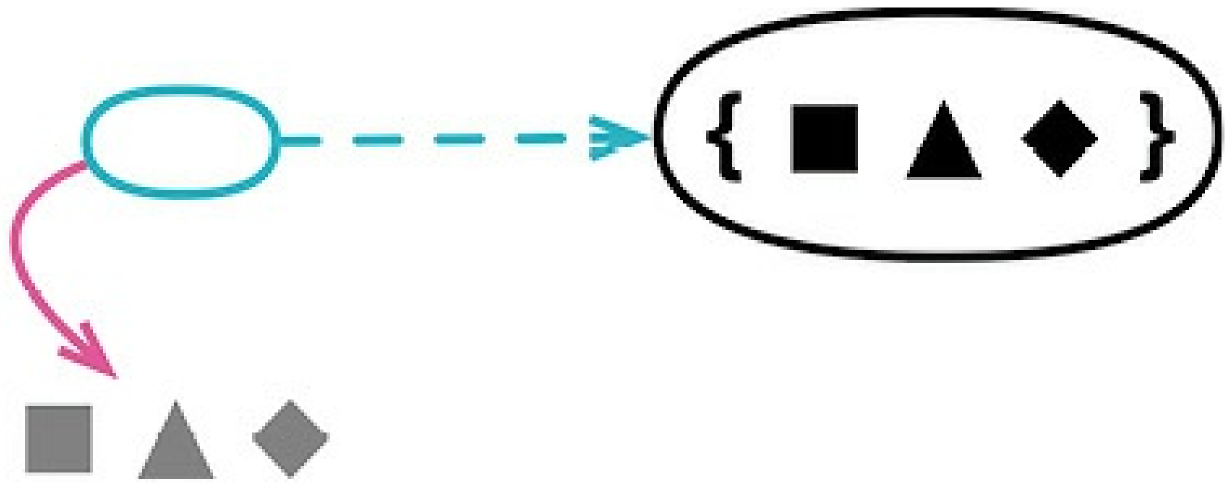
最后，我将zztmp改名为原函数的名字emitPhotoData，完成本次重构。

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
  outStream.write(`<p>location:      ${person.photo.location}
</p>\n`);
}

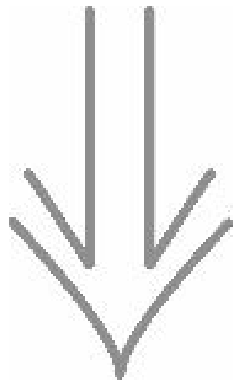
function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}
```

```
function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date:      ${photo.date.toDateString()}
</p>\n`);
}
```

8.5 以函数调用取代内联代码（Replace Inline Code with Function Call）



```
let appliesToMass = false;
for(const s of states) {
  if (s === "MA") appliesToMass = true;
}
```



```
appliesToMass = states.includes("MA");
```

动机

善用函数可以帮助我将相关的行为打包起来，这对于提升代码的表达力大有裨益—— 一个命名良好的函数，本身就能极好地解释代码的用途，使读者不必了解其细节。函数同样有助于消除重复，因为同一段代码我不需要编写两次，每次调用一下函数即可。此外，当我需要修改函数的内部实现时，也不需要四处寻找有没有漏改的相似代码。（当然，我可能需要检查函数的所有调用点，判断它们是否都应该使用新的实现，但通常很少需要这么仔细，即便需要，也总好过四处寻找相似代码。）

如果我见到一些内联代码，它们做的事情仅仅是已有函数的重复，我通常会以一个函数调用取代

内联代码。但有一种情况需要特殊对待，那就是当内联代码与函数之间只是外表相似但其实并无本质联系时。这种情况下，当我改变了函数实现时，并不期望对应内联代码的行为发生改变。判断内联代码与函数之间是否真正重复，从函数名往往可以看出端倪：如果一个函数命名得当，也确实与内联代码做了一样的事，那么这个名字用在内联代码的语境里也应该十分协调；如果函数名显得不协调，可能是因为命名本身就比较糟糕（此时可以运用函数改名（124）来解决），也可能是因为函数与内联代码彼此的用途确实有所不同。若是后者的情况，我就不应该用函数调用取代该内联代码。

我发现，配合一些库函数使用，会使本手法效果更佳，因为我甚至连函数体都不需要自己编写了，库已经提供了相应的函数。

做法

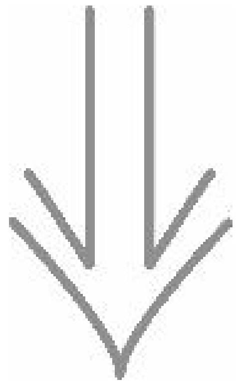
- 将内联代码替代为对一个既有函数的调用。
- 测试。

8.6　移动语句（Slide Statements）

曾用名：合并重复的代码片段（Consolidate Duplicate Conditional Fragments）



```
const pricingPlan = retrievePricingPlan();
const order = retreiveOrder();
let charge;
const chargePerUnit = pricingPlan.unit;
```



```
const pricingPlan = retrievePricingPlan();
const chargePerUnit = pricingPlan.unit;
const order = retreiveOrder();
let charge;
```

动机

让存在关联的东西一起出现，可以使代码更容易理解。如果有几行代码取用了同一个数据结构，那么最好是让它们在一起出现，而不是夹杂在取用其他数据结构的代码中间。最简单的情况下，我只需使用移动语句就可以让它们聚集起来。此外还有一种常见的“关联”，就是关于变量的声明和使用。有人喜欢在函数顶部一口气声明函数用到的所有变量，我个人则喜欢在第一次需要使用变量的地方再声明它。

通常来说，把相关代码搜集到一处，往往是另

一项重构（通常是在提炼函数（106））开始之前的准备工作。相比于仅仅把几行相关的代码移动到一起，将它们提炼到独立的函数往往能起到更好的抽象效果。但如果起先存在关联的代码就没有彼此在一起，那么我也很难应用提炼函数（106）的手法。

做法

- 确定待移动的代码片段应该被搬往何处。仔细检查待移动片段与目的地之间的语句，看看搬移后是否会影响这些代码正常工作。如果会，则放弃这项重构。

  往前移动代码片段时，如果片段中声明了变量，则不允许移动到任何变量的声明语句之前。往后移动代码片段时，如果有语句引用了待移动片段中的变量，则不允许移动到该语句之后。往后移动代码片段时，如果有语句修改了待移动片段中引用的变量，则不允许移动到该语句之后。往后移动代码片段时，如果片段中修改了某些元素，则不允许移动到任何引用

了这些元素的语句之后。

- 剪切源代码片段，粘贴到上一步选定的位置上。
- 测试。

如果测试失败，那么尝试减小移动的步子：要么是减少上下移动的行数，要么是一次搬移更少的代码。

范例

移动代码片段时，通常需要想清楚两件事：本次调整的目标是什么，以及该目标能否达到。第一件事通常取决于代码所在的上下文。最简单的情况是，我希望元素的声明点和使用点互相靠近，因此移动语句的目标便是将元素的声明语句移动到靠近它们的使用处。不过大多数时候，我移动代码的动机都是因为想做另一项重构，比如在应用提炼函数（106）之前先将相关的代码集中到一块，以方便做函数提炼。

确定要把代码移动到哪里之后，我就需要思考第二个问题，也就是此次搬移能否做到的问题。为

此我需要观察待移动的代码，以及移动中间经过的代码段，我得思考这个问题：如果我把代码移动过去，执行次序的不同会不会使代码之间产生干扰，甚至于改变程序的可观测行为？

请观察以下代码片段：

```
 1 const pricingPlan = retrievePricingPlan();
 2 const order = retreiveOrder();
 3 const baseCharge = pricingPlan.base;
 4 let charge;
 5 const chargePerUnit = pricingPlan.unit;
 6 const units = order.units;
 7 let discount;
 8 charge = baseCharge + units * chargePerUnit;
 9 let discountableUnits = Math.max(units - pricingPlan.disco
10 discount = discountableUnits * pricingPlan.discountFactor;
11 if (order.isRepeat) discount += 20;
12 charge = charge - discount;
13 chargeOrder(charge);
```

前七行是变量的声明语句，移动它们通常很简单。假如我想把与处理折扣（discount）相关的代码搬移到一起，那么我可以直接将第7行（`let discount`）移动到第10行上面（`discount = ...`那一行）。因为变量声明没有副作用，也不会引用其他变量，所以我可以很安全地将声明语句往后移动，一直移动到引用`discount`变量的语句之上。此种类型的语句移动也十分常见——当我要提炼函数（106）时，通常得先将相关变量的声明语句搬移过来。

我会再寻找类似的没有副作用的变量声明语句。类似地，我可以毫无障碍地把声明了`order`变量的第2行（`const order = ...`）移动到使用它的第6行（`const units = ...`）上面。

上面搬移变量声明语句之所以顺利，除了因为语句本身没有副作用，还得益于我移动语句时跨过的代码片段同样没有副作用。事实上，对于没有副作用的代码，我几乎可以随心所欲地编排它们的顺序，这也是优秀的程序员都会尽量编写无副作用代码的原因之一。

当然，这里还有一个小细节，那就是我从何得知第2行代码没有副作用呢？我只有深入检查`retrieveOrder()`函数的内部实现，才能真正确保它确实没有副作用（除了检查函数本身，还得检查它内部调用的函数都没有副作用，以及它调用的函数内部调用的函数都没有副作用……一直检查到调用链的底端）。实践中，我编写代码总是尽量遵循命令与查询分离（Command-Query Separation）[mf-cqs]原则，在这个前提下，我可以确定任何有返回值的函数都不存在副作用。但只有在我了解代码库的前提下才如此自信；如果我对代码库还不熟悉，我就得更加小心。但在我自己的编码过程中，我确实总是尽量遵循命令与查询分离的模式，因为它让我一眼就能看清代码有无副作用，而这件事情

真是价值不菲。

如果待移动的代码片段本身有副作用，或者它需要跨越的代码存在副作用，移动它们时就必须加倍小心。我得仔细寻找两个代码片段中间的代码有没有副作用，是不是对执行次序敏感。因此，假设我想将第11行（if(order.isRepeat)...）挪动到段落底部，我会发现行不通，因为中间第12行语句引用了discount变量，而我在第11行中可能改动这个变量；类似地，假设我想将第13行（chargeOrder(charge)）往上搬移，那也是行不通的，因为第13行引用的charge变量在第12行会被修改。不过，如果我想将第8行代码（charge = baseCharge + ...）移动到第9行到第11行中间的任意地方则是可行的，因为这几行都未修改任何变量的状态。

移动代码时，最容易遵守的一条规则是，如果待移动代码片段中引用的变量在另一个代码片段中被修改了，那我就不能安全地将前者移动到后者之后；同样，如果前者会修改后者中引用的变量，也一样不能安全地进行上述移动。但这条规则仅仅作为参考，它也不是绝对的，比如下面这个例子，虽然两个语句都修改了彼此之间的变量，但我仍能安全地调整它们的先后顺序。

```
a = a + 10;
```

```
a = a + 5;
```

但无论如何，要判断一次语句移动是否安全，都意味着我得真正理解代码的工作原理，以及运算符之间的组合方式等。

正因此项重构如此需要关注状态更新，所以我会尽量移除那些会更新元素状态的代码。比如此例中的charge变量，在移动其相关的代码之前，我会先看看是否能对它应用拆分变量（240）手法。

以上的分析都比较简单，没什么难度，因为代码里修改的都是局部变量，局部变量是比较好处理的。但处理更复杂的数据结构时，情况就不同了，判断代码段之间是否存在相互干扰会困难得多。这时测试扮演了重要角色：每次移动代码之后运行测试，看看有没有任何测试失败。如果我的测试覆盖足够全面，我就会对这次重构比较有信心；但如果测试覆盖不够，我就得小心一些了——通常，我会先改善代码的测试，然后再进行重构。

如果移动过后测试失败了，那么意味着我得减小移动的步子，比如一次先移动5行，而不是10行；或者先移动到那些看着比较可能出错的代码上面，但不越过它，看看效果。同时，测试失败也可能是一个征兆，提醒我这次移动可能还不是时候，

可能还需要在别处先做一些其他的工作。

范例：包含条件逻辑的移动

对于拥有条件逻辑的代码，移动手法同样适用。当从条件分支中移走代码时，通常是要消除重复逻辑；将代码移入条件分支时，通常是反过来，有意添加一些重复逻辑。

在下面这个例子中，两个条件分支里都有一个相同的语句：

```
let result;
if (availableResources.length === 0) {
  result = createResource();
  allocatedResources.push(result);
} else {
  result = availableResources.pop();
  allocatedResources.push(result);
}
return result;
```

我可以将这两句重复代码从条件分支中移走，只在`if-else`块的末尾保留一句。

```
let result;
if (availableResources.length === 0) {
  result = createResource();
} else {
```

```
  result = availableResources.pop();
}
allocatedResources.push(result);
return result;
```

这个手法同样可以反过来用，也就是把一个语句分别搬移到不同的条件分支里，这样会在每个条件分支里留下同一段重复的代码。

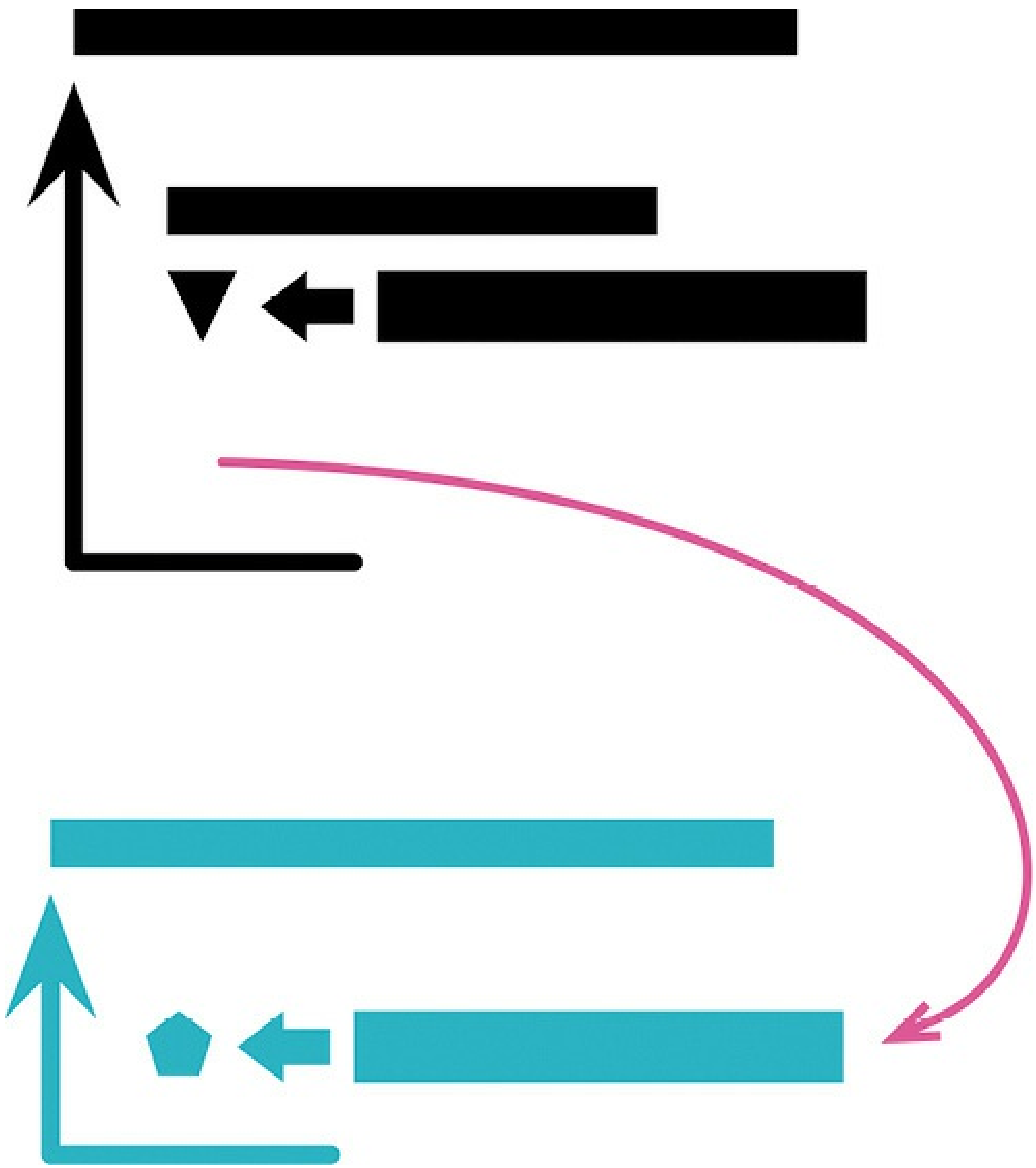
延伸阅读

除了我介绍的这个方法，我还见过一个十分相似的重构手法，名字叫作“交换语句位置”（Swap Statement）[wake-swap]。该手法同样适用于移动相邻的代码片段，只不过它适用的是只有一条语句的片段。你可以把它想成移动语句手法的一个特例，也就是待移动的代码片段以及它所跨过的代码片段，都只有一条语句。我对这项重构手法很感兴趣，毕竟我也一直在强调小步修改——有时甚至小步到于初学重构的人看来都很不可思议的地步。

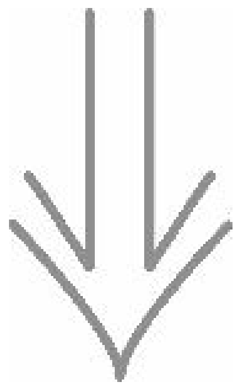
但最后，我还是选择在本重构手法中介绍如何移动范围更大的代码片段，因为我自己平时就是这么做的。我只有在处理大范围的语句移动遇到困难时才会变得小步、一次只移动一条语句，但即便是

这样的困难我也很少遇见。无论如何，当代码过于复杂凌乱时，小步的移动通常会更加顺利。

8.7 拆分循环（Split Loop）



```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```



```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

动机

你常常能见到一些身兼多职的循环，它们一次做了两三件事情，不为别的，就因为这样可以只循

环一次。但如果你在一次循环中做了两件不同的事，那么每当需要修改循环时，你都得同时理解这两件事情。如果能够将循环拆分，让一个循环只做一件事情，那就能确保每次修改时你只需要理解要修改的那块代码的行为就可以了。

拆分循环还能让每个循环更容易使用。如果一个循环只计算一个值，那么它直接返回该值即可；但如果循环做了太多件事，那就只得返回结构型数据或者通过局部变量传值了。因此，一般拆分循环后，我还会紧接着对拆分得到的循环应用提炼函数（106）。

这项重构手法可能让许多程序员感到不安，因为它会迫使你执行两次循环。对此，我一贯的建议也与2.8节里所明确指出的一致：先进行重构，然后再进行性能优化。我得先让代码结构变得清晰，才能做进一步优化；如果重构之后该循环确实成了性能的瓶颈，届时再把拆开的循环合到一起也很容易。但实际情况是，即使处理的列表数据更多一些，循环本身也很少成为性能瓶颈，更何况拆分出循环来通常还使一些更强大的优化手段变得可能。

做法

- 复制一遍循环代码。
- 识别并移除循环中的重复代码，使每个循环只做一件事。
- 测试。

完成循环拆分后，考虑对得到的每个循环应用提炼函数（106）。

范例

下面我以一段循环代码开始。该循环会计算需要支付给所有员工的总薪水（total salary），并计算出最年轻（youngest）员工的年龄。

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}
```

该循环十分简单，但仍然做了两种不同的计算。要拆分这两种计算，我要先复制一遍循环代码。

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}
```

复制过后，我需要将循环中重复的计算逻辑删除，否则就会累加出错误的结果。如果循环中的代码没有副作用，那便可以先留着它们不删除，可惜上述例子并不符合这种情况。

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  ~~if (p.age < youngest) youngest = p.age;~~
  totalSalary += p.salary;
}

for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  ~~totalSalary += p.salary;~~
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}
```

至此，拆分循环这个手法本身的内容就结束了。但本手法的意义不仅在于拆分出循环本身，而且在于它为进一步优化提供了良好的起点——下一

步我通常会寻求将每个循环提炼到独立的函数中。在做提炼之前，我得先用移动语句（223）微调一下代码顺序，将与循环相关的变量先搬移到一起：

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let youngest = people[0] ? people[0].age : Infinity;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}
```

然后，我就可以顺利地应用提炼函数（106）了。

```
return `youngestAge: ${youngestAge()}, totalSalary: ${totalSa

function totalSalary() {
  let totalSalary = 0;
  for (const p of people) {
    totalSalary += p.salary;
  }
  return totalSalary;
}

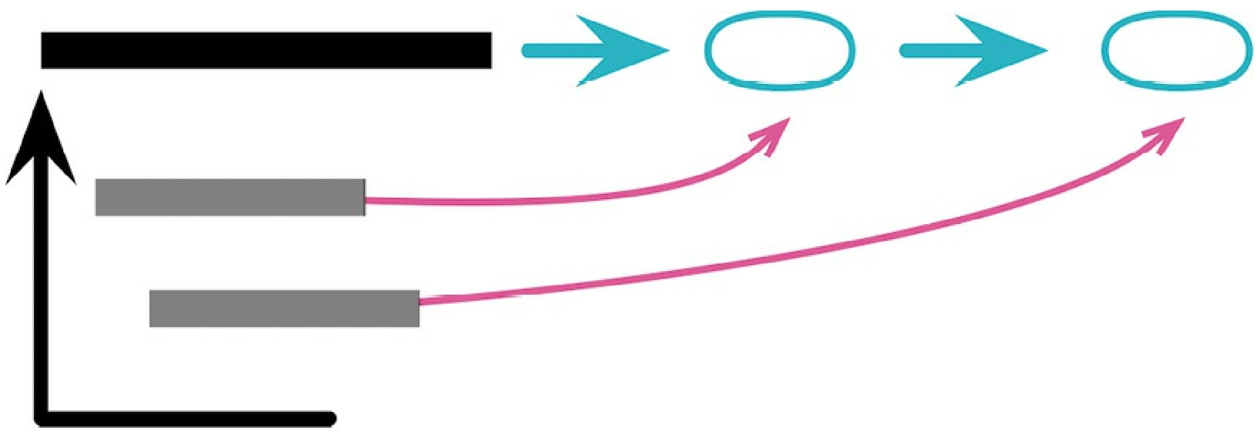
function youngestAge() {
  let youngest = people[0] ? people[0].age : Infinity;
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
  }
  return youngest;
}
```

对于像totalSalary这样的累加计算，我绝少能抵挡得住进一步使用以管道取代循环（231）重构它的诱惑；而对于youngestAge的计算，显然可以用替换算法（195）替之以更好的算法。

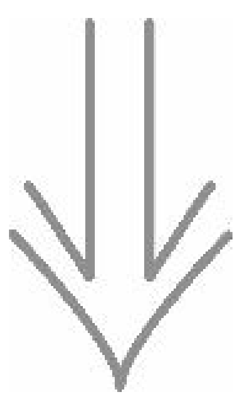
```
return `youngestAge: ${youngestAge()}, totalSalary: ${totalSa

function totalSalary() {
  return people.reduce((total,p) => total + p.salary, 0);
}
function youngestAge() {
  return Math.min(...people.map(p => p.age));
}
```

8.8　以管道取代循环（Replace Loop with Pipeline）



```
const names = [];
for (const i of input) {
  if (i.job === "programmer")
    names.push(i.name);
}
```



```
const names = input
  .filter(i => i.job === "programmer")
  .map(i => i.name)
;
```

动机

与大多数程序员一样，我入行的时候也有人告诉我，迭代一组集合时得使用循环。不过时代在发展，如今越来越多的编程语言都提供了更好的语言结构来处理迭代过程，这种结构就叫作集合管道（collection pipeline）。集合管道[mf-cp]是这样一种技术，它允许我使用一组运算来描述集合的迭代过程，其中每种运算接收的入参和返回值都是一个集合。这类运算有很多种，最常见的则非*map*和*filter*莫属：*map*运算是指用一个函数作用于输入集合的每一个元素上，将集合变换成另外一个集合的过程；*filter*运算是指用一个函数从输入集合中筛选出符合条件的元素子集的过程。运算得到的集合可以供管道的后续流程使用。我发现一些逻辑如果采用集合管道来编写，代码的可读性会更强——我只消从头到尾阅读一遍代码，就能弄清对象在管道中间的变换过程。

做法

- 创建一个新变量，用以存放参与循环过程的集合。

　　也可以简单地复制一个现有的变量赋值给新变量。

- 从循环顶部开始，将循环里的每一块行为依次搬移出来，在上一步创建的集合变量上用一种管道运算替代之。每次修改后运行测试。
- 搬移完循环里的全部行为后，将循环整个删除。

　　如果循环内部通过累加变量来保存结果，那么移除循环后，将管道运算的最终结果赋值给该累加变量。

范例

在这个例子中，我们有一个CSV文件，里面存有各个办公室（office）的一些数据。

```
office, country, telephone
Chicago, USA, +1 312 373 1000
Beijing, China, +86 4008 900 505
Bangalore, India, +91 80 4064 9570
Porto Alegre, Brazil, +55 51 3079 3550
Chennai, India, +91 44 660 44766

... (more data follows)
```

下面这个`acquireData`函数的作用是从数据中筛选出印度的所有办公室，并返回办公室所在的城市（city）信息和联系电话（telephone number）。

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  for (const line of lines) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
result.push({city: record[0].trim(), phone: record[2].trim()}
    }
  }
  return result;
}
```

这个循环略显复杂，我希望能用一组管道操作来替换它。

第一步是先创建一个独立的变量，用来存放参与循环过程的集合值。

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  const loopItems = lines
  for (const line of loopItems) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {

result.push({city: record[0].trim(), phone: record[2].trim()}
    }
  }
  return result;
}
```

循环第一部分的作用是在忽略CSV文件的第一行数据。这其实是一个切片（slice）操作，因此我先从循环中移除这部分代码，并在集合变量的声明后面新增一个对应的`slice`运算来替代它。

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  const loopItems = lines
```

```
        .slice(1);
  for (const line of loopItems) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {

result.push({city: record[0].trim(), phone: record[2].trim()}
    }
  }
  return result;
}
```

从循环中删除代码还有一个好处，那就是`firstLine`这个控制变量也可以一并移除了——无论何时，删除控制变量总能使我身心愉悦。

该循环的下一个行为是要移除数据中的所有空行。这同样可用一个过滤（filter）运算替代之。

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        ;
  for (const line of loopItems) {
    if (line.trim() === "") continue;
    const  record = line.split(",");
    if (record[1].trim() === "India") {

result.push({city: record[0].trim(), phone: record[2].trim()}
    }
  }
  return result;
}
```

编写管道运算时，我喜欢让结尾的分号单占一行，这样方便调整管道的结构。

接下来是将数据的一行转换成数组，这明显可以用一个map运算替代。然后我们还发现，原来的record命名其实有误导性，它没有体现出“转换得到的结果是数组”这个信息，不过既然现在还在做其他重构，先不动它会比较安全，回头再为它改名也不迟。

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        ;
  for (const line of loopItems) {
    const record = line;.split(",");
    if (record[1].trim() === "India") {

result.push({city: record[0].trim(), phone: record[2].trim()}
    }
  }
  return result;
}
```

然后又是一个过滤（filter）操作，只从结果中筛选出印度办公室的记录。

```
function acquireData(input) {
  const lines = input.split("\n");
```

```
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        ;
  for (const line of loopItems) {
    const record = line;
    if (record[1].trim() === "India") {

result.push({city: record[0].trim(), phone: record[2].trim()}
    }
  }
  return result;
}
```

最后再把结果映射（map）成需要的记录格式：

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        .map(record => ({city: record[0].trim(), phone: record
        ;
  for (const line of loopItems) {
    const record = line;
    result.push(line);
  }
  return result;
}
```

现在，循环剩余的唯一作用就是对累加变量赋

值了。我可以将上面管道产出的结果赋值给该累加变量，然后删除整个循环：

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        .map(record => ({city: record[0].trim(), phone: record
        ;
  for (const line of loopItems) {
    const record = line;
    result.push(line);
  }
  return result;
}
```

以上就是本手法的全部精髓所在了。不过后续还有些清理工作可做：我内联了result变量，为一些函数变量改名，最后还对代码进行布局，让它读起来更像个表格。

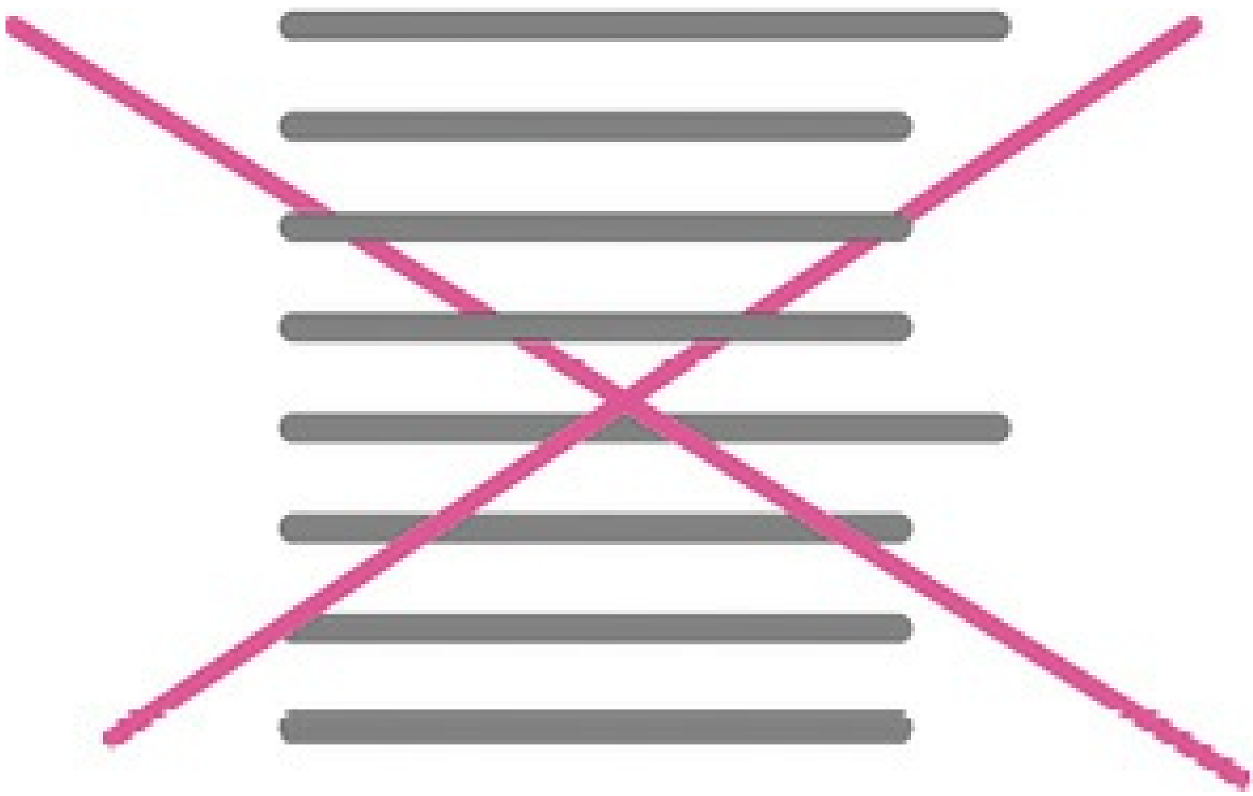
```
function acquireData(input) {
  const lines = input.split("\n");
  return lines
        .slice (1)
        .filter (line => line.trim() !== "")
        .map   (line => line.split(","))
        .filter (fields => fields[1].trim() === "India")
        .map   (fields => ({city: fields[0].trim(), phone: fie
        ;
}
```

我还想过是否要内联`lines`变量，但我感觉它还算能解释该行代码的意图，因此我还是将它留在了原处。

延伸阅读

如果想了解更多用集合管道替代循环的案例，可以参考我的文章“Refactoring with Loops and Collection Pipelines”[mf-ref-pipe]。

8.9 移除死代码（Remove Dead Code）



```
if(false) {
  doSomethingThatUsedToMatter();
}
```

动机

事实上，我们部署到生产环境甚至是用户设备上的代码，从来未因代码量太大而产生额外费用。就算有几行用不上的代码，似乎也不会因此拖慢系统速度，或者占用过多的内存，大多数现代的编译器还会自动将无用的代码移除。但当你尝试阅读代码、理解软件的运作原理时，无用代码确实会带来很多额外的思维负担。它们周围没有任何警示或标记能告诉程序员，让他们能够放心忽略这段函数，因为已经没有任何地方使用它了。当程序员花费了许多时间，尝试理解它的工作原理时，却发现无论怎么修改这段代码都无法得到期望的输出。

一旦代码不再被使用，我们就该立马删除它。有可能以后又会需要这段代码，但我从不担心这种情况；就算真的发生，我也可以从版本控制系统里再次将它翻找出来。如果我真的觉得日后它极有可

能再度启用，那还是要删掉它，只不过可以在代码里留一段注释，提一下这段代码的存在，以及它被移除的那个提交版本号——但老实讲，我已经记不得我上次撰写这样的注释是什么时候了，当然也未曾因为不写而感到过遗憾。

在以前，业界对于死代码的处理态度多是注释掉它。在版本控制系统还未普及、用起来又不太方便的年代，这样做有其道理；但现在版本控制系统已经相当普及。如今哪怕是一个极小的代码库我都会把它放进版本控制，这些无用代码理应可以放心清理了。

做法

- 如果死代码可以从外部直接引用，比如它是一个独立的函数时，先查找一下还有无调用点。
- 将死代码移除。
- 测试。

第9章　重新组织数据

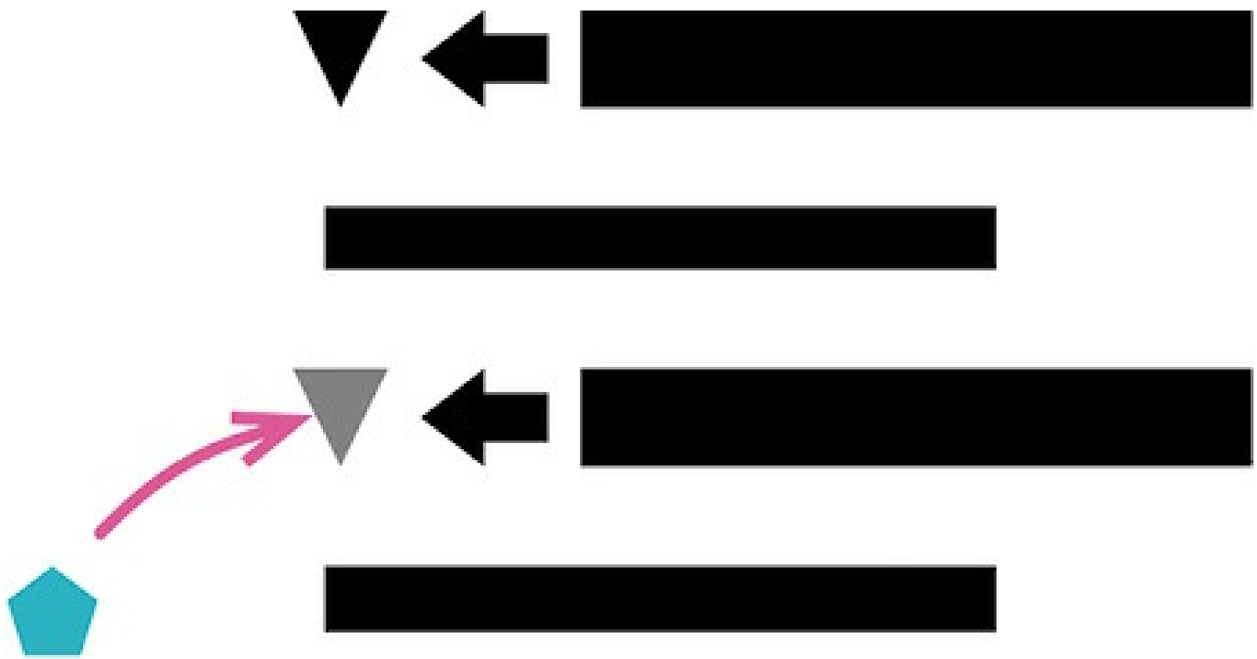
数据结构在程序中扮演着重要的角色，所以毫不意外，我有一组重构手法专门用于数据结构的组织。将一个值用于多个不同的用途，这就是催生混乱和bug的温床。所以，一旦看见这样的情况，我就会用拆分变量（240）将不同的用途分开。和其他任何程序元素一样，给变量起个好名字不容易但又非常重要，所以我常会用到变量改名（137）。但有些多余的变量最好是彻底消除掉，比如通过以查询取代派生变量（248）。

引用和值的混淆经常会造成问题，所以我会用将引用对象改为值对象（252）和将值对象改为引用对象（256）在两者之间切换。

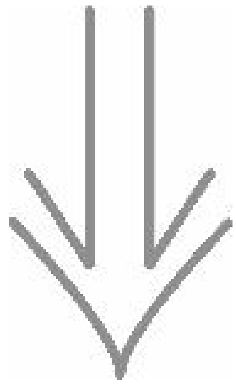
9.1　拆分变量（Split Variable）

曾用名：移除对参数的赋值（Remove Assignments to Parameters）

曾用名：分解临时变量（Split Temp）



```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```



```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

动机

变量有各种不同的用途，其中某些用途会很自然地导致临时变量被多次赋值。“循环变量”和“结果收集变量”就是两个典型例子：循环变量（loop variable）会随循环的每次运行而改变（例如`for（let i=0; i<10; i++）`语句中的`i`）；结果收集变量（collecting variable）负责将“通过整个函数的运算”而构成的某个值收集起来。

除了这两种情况，还有很多变量用于保存一段冗长代码的运算结果，以便稍后使用。这种变量应该只被赋值一次。如果它们被赋值超过一次，就意

味它们在函数中承担了一个以上的责任。如果变量承担多个责任，它就应该被替换（分解）为多个变量，每个变量只承担一个责任。同一个变量承担两件不同的事情，会令代码阅读者糊涂。

做法

- 在待分解变量的声明及其第一次被赋值处，修改其名称。

    如果稍后的赋值语句是“`i=i+`某表达式形式”，意味着这是一个结果收集变量，就不要分解它。结果收集变量常用于累加、字符串拼接、写入流或者向集合添加元素。

- 如果可能的话，将新的变量声明为不可修改。
- 以该变量的第二次赋值动作为界，修改此前对该变量的所有引用，让它们引用新变量。
- 测试。
- 重复上述过程。每次都在声明处对变量改名，并修改下次赋值之前的引用，直至到达最后一

处赋值。

范例

下面范例中我要计算一个苏格兰布丁运动的距离。在起点处，静止的苏格兰布丁会受到一个初始力的作用而开始运动。一段时间后，第二个力作用于布丁，让它再次加速。根据牛顿第二定律，我可以这样计算布丁运动的距离：

```
function distanceTravelled (scenario, time) {
  let result;
  let acc = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * acc * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = acc * scenario.delay;

acc = (scenario.primaryForce + scenario.secondaryForce) / sce

result += primaryVelocity * secondaryTime + 0.5 * acc * secon
  }
  return result;
}
```

真是个丑陋的小东西。注意观察此例中的acc变量是如何被赋值两次的。acc变量有两个责任：第一是保存第一个力造成的初始加速度；第二是保存两个力共同造成的加速度。这就是我想要分解的

东西。

在尝试理解变量被如何使用时，如果编辑器能高亮显示一个符号（symbol）在函数内或文件内出现的所有位置，会相当便利。大部分现代编辑器都可以轻松做到这一点。

首先，我在函数开始处修改这个变量的名称，并将新变量声明为const。接着，我把新变量声明之后、第二次赋值之前对acc变量的所有引用，全部改用新变量。最后，我在第二次赋值处重新声明acc变量：

```
function distanceTravelled (scenario, time) {
  let result;

const primaryAcceleration = scenario.primaryForce / scenario.
  let primaryTime = Math.min(time, scenario.delay);

result = 0.5 * primaryAcceleration * primaryTime * primaryTim
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {

let primaryVelocity = primaryAcceleration * scenario.delay;

let acc = (scenario.primaryForce + scenario.secondaryForce) /

result += primaryVelocity * secondaryTime + 0.5 * acc * secon
  }
  return result;
}
```

新变量的名称指出，它只承担原先acc变量的第一个责任。我将它声明为const，确保它只被赋值一次。然后，我在原先acc变量第二次被赋值处重新声明acc。现在，重新编译并测试，一切都应该没有问题。

然后，我继续处理acc变量的第二次赋值。这次我把原先的变量完全删掉，代之以一个新变量。新变量的名称指出，它只承担原先acc变量的第二个责任：

```
function distanceTravelled (scenario, time) {
  let result;

const primaryAcceleration = scenario.primaryForce / scenario.
  let primaryTime = Math.min(time, scenario.delay);

result = 0.5 * primaryAcceleration * primaryTime * primaryTim
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {

let primaryVelocity = primaryAcceleration * scenario.delay;

const secondaryAcceleration = (scenario.primaryForce + scenar
    result += primaryVelocity * secondaryTime +

0.5 * secondaryAcceleration * secondaryTime * secondaryTime;
  }
  return result;
}
```

现在，这段代码肯定可以让你想起更多其他重构手法。尽情享受吧。（我敢保证，这比吃苏格兰布丁强多了——你知道他们都在里面放了些什么东

西吗？1 ）

范例：对输入参数赋值

另一种情况是，变量是以输入参数的形式声明又在函数内部被再次赋值，此时也可以考虑拆分变量。例如，下列代码：

```
function discount (inputValue, quantity) {
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

这里的inputValue有两个用途：它既是函数的输入，也负责把结果带回给调用方。（由于JavaScript的参数是按值传递的，所以函数内部对inputValue的修改不会影响调用方。）

在这种情况下，我就会对inputValue变量做拆分。

```
function discount (originalInputValue, quantity) {
  let inputValue = originalInputValue;
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

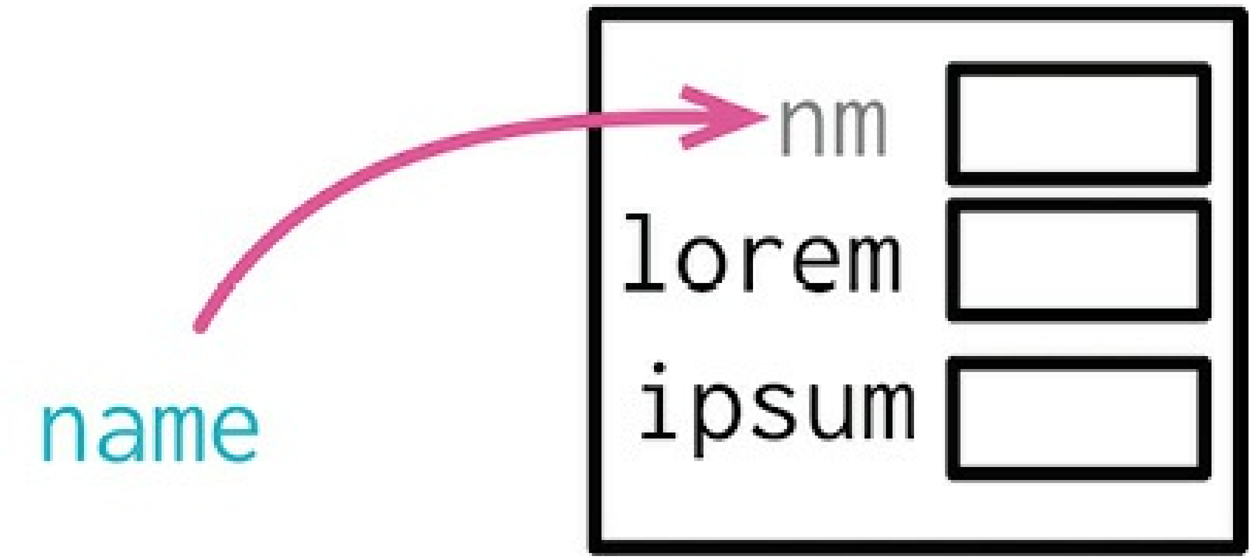
然后用变量改名（137）给两个变量换上更好的名字。

```
function discount (inputValue, quantity) {
  let result = inputValue;
  if (inputValue > 50) result = result - 2;
  if (quantity > 100) result = result - 1;
  return result;
}
```

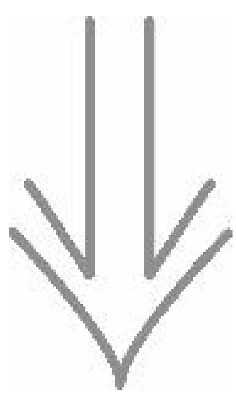
我修改了第二行代码，把`inputValue`作为判断条件的基准数据。虽说这里用`inputValue`还是`result`效果都一样，但在我看来，这行代码的含义是“根据原始输入值做判断，然后修改结果值”，而不是“根据当前结果值做判断”——尽管两者的效果恰好一样。

1苏格兰布丁（haggis）是一种苏格兰菜，把羊心等内脏装在羊胃里煮成。由于它被羊胃包成一个球体，因此可以像球一样踢来踢去，这就是本例的由来。“把羊心装在羊胃里煮成……”，呃，有些人难免对这道菜恶心，Martin Fowler想必是其中之一。——译者注

9.2　字段改名（Rename Field）




```
class Organization {
  get name() {...}
}
```



```
class Organization {
```

```
  get title() {...}
}
```

动机

命名很重要，对于程序中广泛使用的记录结构，其中字段的命名格外重要。数据结构对于帮助阅读者理解特别重要。多年以前，Fred Brooks就说过："只给我看你的工作流程却隐藏表单，我将仍然一头雾水。但是如果你给我展示表单，或许不需要流程图，就能柳暗花明。"现在已经不太有人画流程图了，不过道理还是一样的。数据结构是理解程序行为的关键。

既然数据结构如此重要，就很有必要保持它们的整洁。一如既往地，我在一个软件上做的工作越多，对数据的理解就越深，所以很有必要把我加深的理解融入程序中。

记录结构中的字段可能需要改名，类的字段也一样。在类的使用者看来，取值和设值函数就等于是字段。对这些函数的改名，跟裸记录结构的字段改名一样重要。

做法

- 如果记录的作用域较小，可以直接修改所有该字段的代码，然后测试。后面的步骤就都不需要了。
- 如果记录还未封装，请先使用封装记录（162）。
- 在对象内部对私有字段改名，对应调整内部访问该字段的函数。
- 测试。
- 如果构造函数的参数用了旧的字段名，运用改变函数声明（124）将其改名。
- 运用函数改名（124）给访问函数改名。

范例：给字段改名

我们从一个常量开始。

```
const organization = {name: "Acme Gooseberries", country: "GB
```

我想把`name`改名为`title`。这个对象被很多地

方使用，有些代码会更新name字段。所以我首先要用封装记录（162）把这个记录封装起来。

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()  {return this._name;}
  set name(aString) {this._name = aString;}
  get country()  {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}

const organization = new Organization({name: "Acme Gooseberri
```

现在，记录结构已经被封装成类。在对字段改名时，有4个地方需要留意：取值函数、设值函数、构造函数以及内部数据结构。这听起来似乎是增加了重构的工作量，但现在我可以分别小步修改这4处，而不必一次修改所有地方，所以其实是降低了重构的难度。小步修改就意味着每一步出错的可能性大大减小，因此会省掉很多工作量——如果我从不犯错，小步前进不会节省工作量；但“从不犯错”这样的梦，我很久以前就已经不做了。

由于已经把输入数据复制到内部数据结构中，现在我需要将这两个数据结构区分开，以便各自单独处理。我可以另外定义一个字段，修改构造函数和访问函数，令其使用新字段。

class Organization...

```
class Organization {
  constructor(data) {
    this._title = data.name;
    this._country = data.country;
  }
  get name()  {return this._title;}
  set name(aString) {this._title = aString;}
  get country()  {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

接下来我就可以在构造函数中使用title字段。

class Organization...

```
class Organization {
  constructor(data) {

this._title = (data.title !== undefined) ? data.title : data.
    this._country = data.country;
  }
  get name()       {return this._title;}
  set name(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

现在，构造函数的调用者既可以使用name也可以使用title（后者的优先级更高）。我会逐一查

看所有调用构造函数的地方，将它们改为使用新名字。

```
const organization = new Organization({title: "Acme Gooseberr
```

全部修改完成后，就可以在构造函数中去掉对name的支持，只使用title。

class Organization...

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get name()       {return this._title;}
  set name(aString) {this._title = aString;}
  get country()     {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

现在构造函数和内部数据结构都已经在使用新名字了，接下来我就可以给访问函数改名。这一步很简单，只要对每个访问函数运用函数改名（124）就行了。

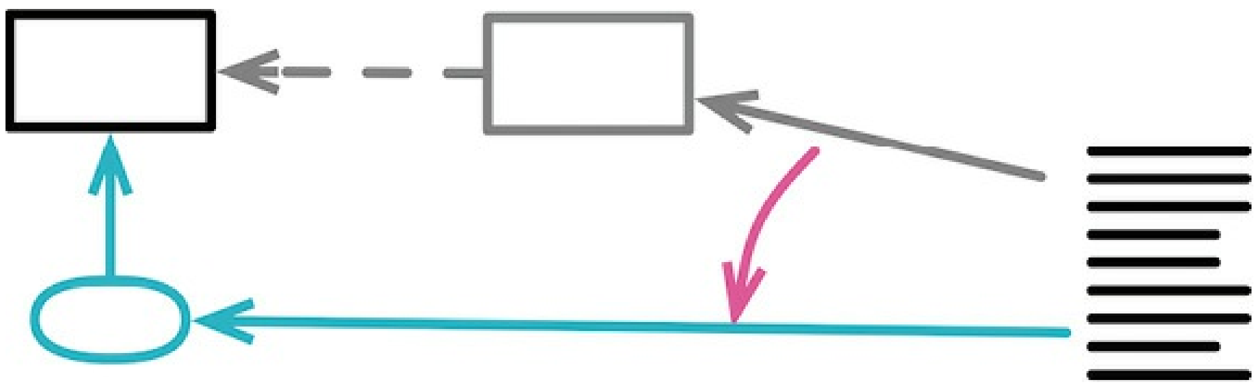
class Organization...

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get title()  {return this._title;}
  set title(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

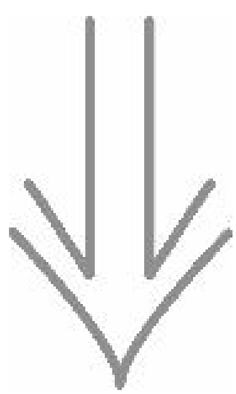
上面展示的重构过程，是本重构手法最重量级的做法，只有对广泛使用的数据结构才用得上。如果该数据结构只在较小的范围（例如单个函数）中用到，我可能可以一步到位地完成所有改名动作，不需要提前做封装。何时需要用上全套重量级做法，这由你自己判断——如果在重构过程中破坏了测试，我通常会视之为一个信号，说明我需要改用更渐进的方式来重构。

有些编程语言允许将数据结构声明为不可变。在这种情况下，我可以把旧字段的值复制到新名字下，逐一修改使用方代码，然后删除旧字段。对于可变的数据结构，重复数据会招致灾难；而不可变的数据结构则没有这些麻烦。这也是大家愿意使用不可变数据的原因。

9.3　以查询取代派生变量（Replace Derived Variable with Query）



```
get discountedTotal() {return this._discountedTotal;}
set discount(aNumber) {
  const old = this._discount;
  this._discount = aNumber;
  this._discountedTotal += old - aNumber;
}
```



```
get discountedTotal() {return this._baseTotal - this._discoun
set discount(aNumber) {this._discount = aNumber;}
```

动机

可变数据是软件中最大的错误源头之一。对数据的修改常常导致代码的各个部分以丑陋的形式互相耦合：在一处修改数据，却在另一处造成难以发现的破坏。很多时候，完全去掉可变数据并不现实，但我还是强烈建议：尽量把可变数据的作用域限制在最小范围。

有些变量其实可以很容易地随时计算出来。如果能去掉这些变量，也算朝着消除可变性的方向迈出了一大步。计算常能更清晰地表达数据的含义，而且也避免了“源数据修改时忘了更新派生变量”的错误。

有一种合理的例外情况：如果计算的源数据是不可变的，并且我们可以强制要求计算的结果也是不可变的，那么就不必重构消除计算得到的派生变量。因此，“根据源数据生成新数据结构”的变换操作可以保持不变，即便我们可以将其替换为计算操作。实际上，这是两种不同的编程风格：一种是对

象风格，把一系列计算得出的属性包装在数据结构中；另一种是函数风格，将一个数据结构变换为另一个数据结构。如果源数据会被修改，而你必须负责管理派生数据结构的整个生命周期，那么对象风格显然更好。但如果源数据不可变，或者派生数据用过即弃，那么两种风格都可行。

做法

- 识别出所有对变量做更新的地方。如有必要，用拆分变量（240）分割各个更新点。
- 新建一个函数，用于计算该变量的值。
- 用引入断言（302）断言该变量和计算函数始终给出同样的值。

如有必要，用封装变量（132）将这个断言封装起来。

- 测试。
- 修改读取该变量的代码，令其调用新建的函数。

- 测试。
- 用移除死代码（237）去掉变量的声明和赋值。

范例

下面这个例子虽小，却完美展示了代码的丑陋。

class ProductionPlan...

```
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

丑与不丑，全在观者。我看到的丑陋之处是重复——不是常见的代码重复，而是数据的重复。如果我要对生产计划（production plan）做调整（adjustment），不光要把调整的信息保存下来，还要根据调整信息修改一个累计值——后者完全可以即时计算，而不必每次更新。

但我是个谨慎的人。“可以即时计算”只是我的猜想——我可以用引入断言（302）来验证这个猜

想。

class ProductionPlan...

```
get production() {
  assert(this._production === this.calculatedProduction);
  return this._production;
}

get calculatedProduction() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

放上这个断言之后，我会运行测试。如果断言没有失败，我就可以不再返回该字段，改为返回即时计算的结果。

class ProductionPlan...

```
get production() {
  assert(this._production === this.calculatedProduction);
  return this.calculatedProduction;
}
```

然后用内联函数（115）把计算逻辑内联到production函数内。

class ProductionPlan...

```
get production() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

再用移除死代码（237）扫清使用旧变量的地方。

class ProductionPlan...

```
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

范例：不止一个数据来源

上面的例子处理得轻松愉快，因为`production`的值很明显只有一个来源。但有时候，累计值会受到多个数据来源的影响。

class ProductionPlan...

```
constructor (production) {
  this._production = production;
  this._adjustments = [];
}
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

如果照上面的方式运用引入断言（302），只要`production`的初始值不为0，断言就会失败。

不过我还是可以替换派生数据，只不过必须先运用拆分变量（240）。

```
constructor (production) {
  this._initialProduction = production;
  this._productionAccumulator = 0;
  this._adjustments = [];
}
get production() {

return this._initialProduction + this._productionAccumulator;
}
```

现在我就可以使用引入断言（302）。

class ProductionPlan...

```
get production() {

assert(this._productionAccumulator === this.calculatedProduct

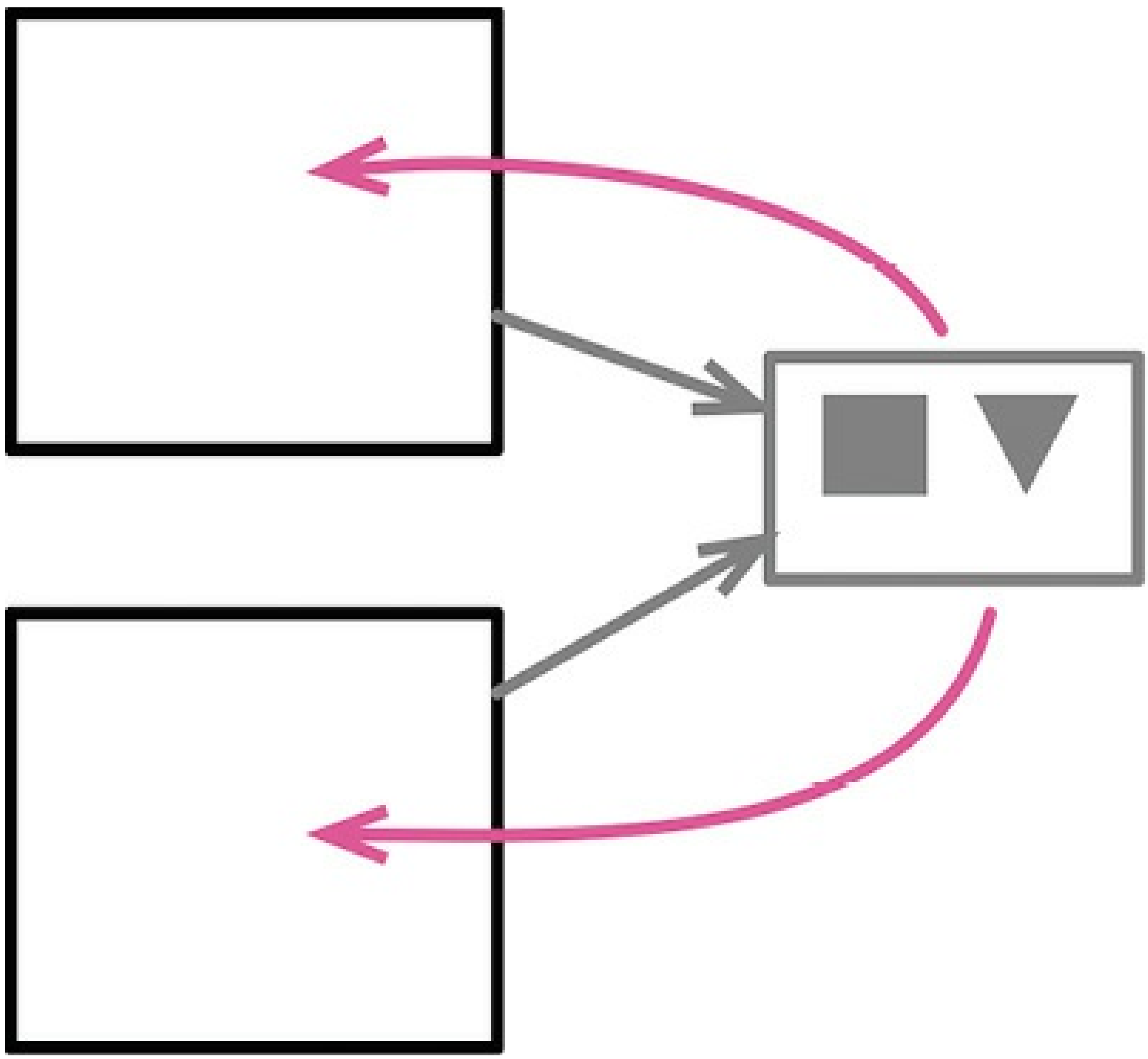
return this._initialProduction + this._productionAccumulator;
}

get calculatedProductionAccumulator() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

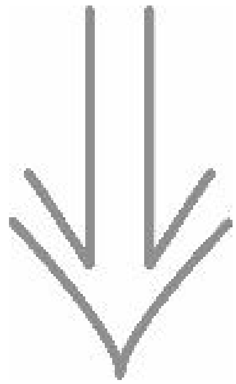
接下来的步骤就跟前一个范例一样了。不过我会更愿意保留calculatedProduction Accumulator这个属性，而不把它内联消去。

9.4　将引用对象改为值对象（Change Reference to Value）

反向重构：将值对象改为引用对象（256）



```
class Product {
  applyDiscount(arg) {this._price.amount -= arg;}
```



```
class Product {
  applyDiscount(arg) {
    this._price = new Money(this._price.amount - arg, this._p
  }
```

动机

在把一个对象（或数据结构）嵌入另一个对象时，位于内部的这个对象可以被视为引用对象，也可以被视为值对象。两者最明显的差异在于如何更新内部对象的属性：如果将内部对象视为引用对象，在更新其属性时，我会保留原对象不动，更新内部对象的属性；如果将其视为值对象，我就会替换整个内部对象，新换上的对象会有我想要的属性值。

如果把一个字段视为值对象，我可以把内部对象的类也变成值对象[mf-vo]。值对象通常更容易理解，主要因为它们是不可变的。一般说来，不可变的数据结构处理起来更容易。我可以放心地把不可变的数据值传给程序的其他部分，而不必担心对象中包装的数据被偷偷修改。我可以在程序各处复制值对象，而不必操心维护内存链接。值对象在分布式系统和并发系统中尤为有用。

值对象和引用对象的区别也告诉我，何时不应该使用本重构手法。如果我想在几个对象之间共享一个对象，以便几个对象都能看见对共享对象的修改，那么这个共享的对象就应该是引用。

做法

- 检查重构目标是否为不可变对象，或者是否可修改为不可变对象。
- 用移除设值函数（331）逐一去掉所有设值函数。
- 提供一个基于值的相等性判断函数，在其中使用值对象的字段。

大多数编程语言都提供了可覆写的相等性判断函数。通常你还必须同时覆写生成散列码的函数。

范例

设想一个代表“人”的Person类，其中包含一个代表“电话号码”的Telephone Number对象。

class Person...

constructor() {

```
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}

get                                         officeAreaCode()
{return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg
get officeNumber()   {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

class TelephoneNumber...

```
get areaCode()    {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}

get number()    {return this._number;}
set number(arg) {this._number = arg;}
```

代码的当前状态是提炼类（182）留下的结果：从前拥有电话号码信息的Person类仍然有一些函数在修改新对象的属性。趁着还只有一个指向新类的引用，现在是时候使用将引用对象改为值对象将其变成值对象。

我需要做的第一件事是把TelephoneNumber类变成不可变的。对它的字段运用移除设值函数（331）。移除设值函数（331）的第一步是，用改变函数声明（124）把这两个字段的初始值加到构造函数中，并迫使构造函数调用设值函数。

class TelephoneNumber...

```
constructor(areaCode, number) {
  this._areaCode = areaCode;
  this._number = number;
}
```

然后我会逐一查看设值函数的调用者，并将其改为重新赋值整个对象。先从“地区代码”（area

code）开始。

class Person...

```
get officeAreaCode()    {return this._telephoneNumber.areaCod
set officeAreaCode(arg) {

this._telephoneNumber = new TelephoneNumber(arg, this.officeN
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

对于其他字段，重复上述步骤。

class Person...

```
get officeAreaCode()    {return this._telephoneNumber.areaCod
set officeAreaCode(arg) {
  this._telephoneNumber = new TelephoneNumber(arg, this.offic
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {

this._telephoneNumber = new TelephoneNumber(this.officeAreaCo
}
```

现在，TelephoneNumber已经是不可变的类，可以将其变成真正的值对象了。是不是真正的值对象，要看是否基于值判断相等性。在这个领域中，

JavaScript做得不好：语言和核心库都不支持将“基于引用的相等性判断”换成“基于值的相等性判断”。我唯一能做的就是创建自己的`equals`函数。

class TelephoneNumber...

```
equals(other) {
  if (!(other instanceof TelephoneNumber)) return false;
  return this.areaCode === other.areaCode &&
    this.number === other.number;
}
```

对其进行测试很重要：

```
it('telephone equals', function() {
  assert(       new TelephoneNumber("312", "555-0142")
        .equals(new TelephoneNumber("312", "555-0142")));
});
```

这段测试代码用了不寻常的格式，是为了帮助读者一眼看出上下两次构造函数调用完全一样。

我在这个测试中创建了两个各自独立的对象，并验证它们相等。

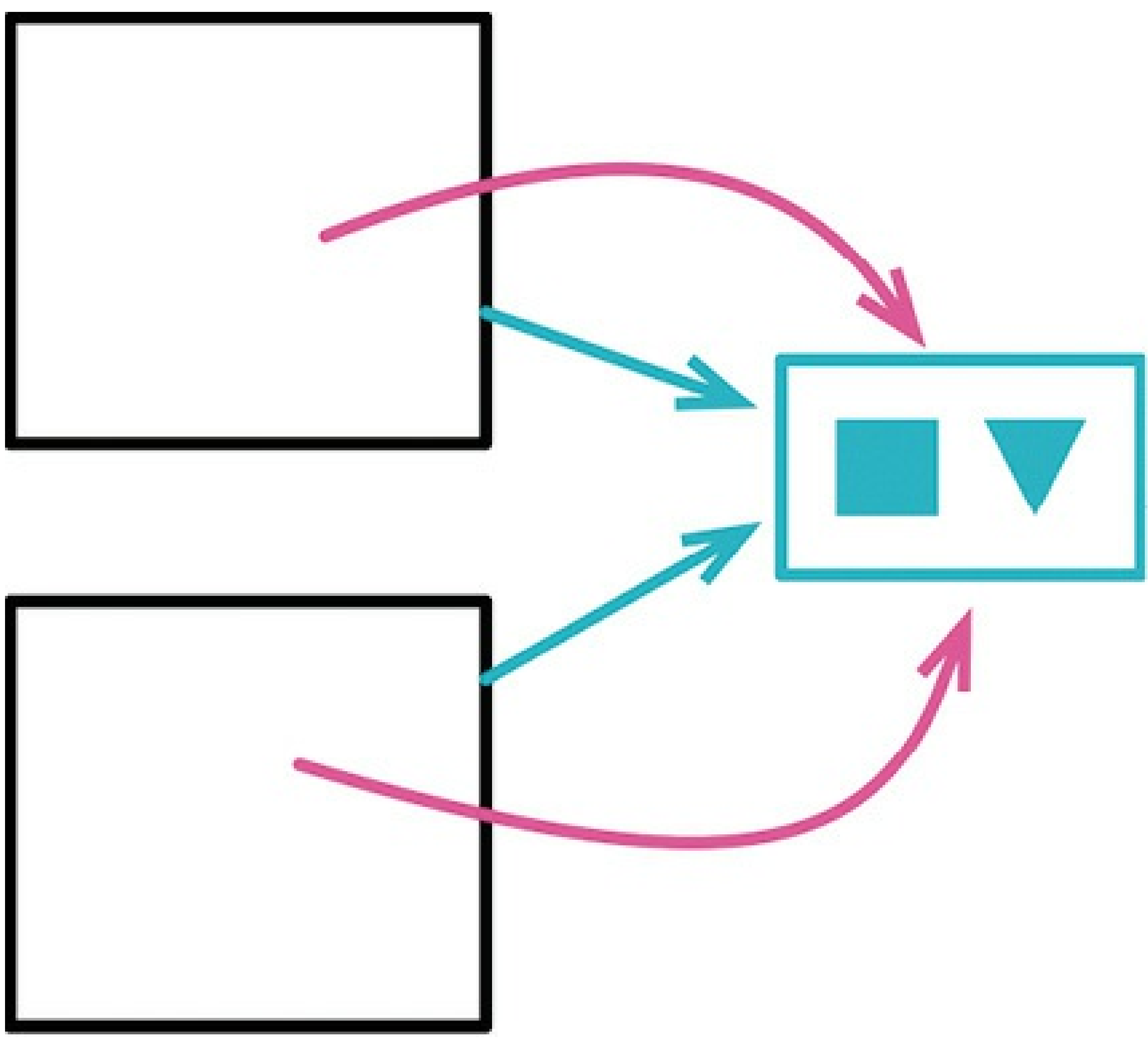
在大多数面向对象语言中，内置的相等性

判断方法可以被覆写为基于值的相等性判断。在Ruby中，我可以覆写==运算符；在Java中，我可以覆写Object.equals()方法。在覆写相等性判断的同时，我通常还需要覆写生成散列码的方法（例如Java中的Object.hashCode()方法），以确保用到散列码的集合在使用值对象时一切正常。

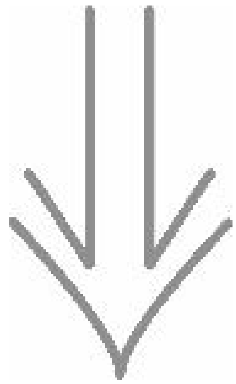
如果有多个客户端使用了TelephoneNumber对象，重构的过程还是一样，只是在运用移除设值函数（331）时要修改多处客户端代码。另外，有必要添加几个测试，检查电话号码不相等以及与非电话号码和null值比较相等性等情况。

9.5　将值对象改为引用对象（Change Value to Reference）

反向重构：将引用对象改为值对象（252）



```
let customer = new Customer(customerData);
```



```
let customer = customerRepository.get(customerData.id);
```

动机

一个数据结构中可能包含多个记录，而这些记录都关联到同一个逻辑数据结构。例如，我可能会读取一系列订单数据，其中有多条订单属于同一个顾客。遇到这样的共享关系时，既可以把顾客信息作为值对象来看待，也可以将其视为引用对象。如果将其视为值对象，那么每份订单数据中都会复制顾客的数据；而如果将其视为引用对象，对于一个顾客，就只有一份数据结构，会有多个订单与之关联。

如果顾客数据永远不修改，那么两种处理方式都合理。把同一份数据复制多次可能会造成一点困扰，但这种情况也很常见，不会造成太大问题。过

多的数据复制有可能会造成内存占用的问题，但就跟所有性能问题一样，这种情况并不常见。

如果共享的数据需要更新，将其复制多份的做法就会遇到巨大的困难。此时我必须找到所有的副本，更新所有对象。只要漏掉一个副本没有更新，就会遭遇麻烦的数据不一致。这种情况下，可以考虑将多份数据副本变成单一的引用，这样对顾客数据的修改就会立即反映在该顾客的所有订单中。

把值对象改为引用对象会带来一个结果：对于一个客观实体，只有一个代表它的对象。这通常意味着我会需要某种形式的仓库，在仓库中可以找到所有这些实体对象。只为每个实体创建一次对象，以后始终从仓库中获取该对象。

做法

- 为相关对象创建一个仓库（如果还没有这样一个仓库的话）。
- 确保构造函数有办法找到关联对象的正确实例。
- 修改宿主对象的构造函数，令其从仓库中获取关联对象。每次修改后执行测试。

范例

我将从一个代表“订单”的Order类开始，其实例对象可从一个JSON文件创建。用来创建订单的数据中有一个顾客（customer）ID，我们用它来进一步创建Customer对象。

class Order...

```
constructor(data) {
  this._number = data.number;
  this._customer = new Customer(data.customer);
  // load other data
}
get customer() {return this._customer;}
```

class Customer...

```
constructor(id) {
  this._id = id;
}
get id() {return this._id;}
```

以这种方式创建的Customer对象是值对象。如

果有5个订单都属于ID为123的顾客，就会有5个各自独立的Customer对象。对其中一个所做的修改，不会反映在其他几个对象身上。如果我想增强Customer对象，例如从客户服务获取到了更多关于顾客的信息，我必须用同样的数据更新所有5个对象。重复的对象总是会让我紧张——用多个对象代表同一个实体（例如一名顾客），这会招致混乱。如果Customer对象是可变的，问题就更加严重，因为各个对象之间的数据可能不一致。

如果我想每次都使用同一个Customer对象，那么就需要有一个地方存储这个对象。每个应用程序中，存储实体的地方会各有不同，在最简单的情况下，我会使用一个仓库对象[mf-repos]。

```
let _repositoryData;

export function initialize() {
  _repositoryData = {};
  _repositoryData.customers = new Map();
}

export function registerCustomer(id) {
  if (! _repositoryData.customers.has(id))
    _repositoryData.customers.set(id, new Customer(id));
  return findCustomer(id);
}

export function findCustomer(id) {
  return _repositoryData.customers.get(id);
}
```

仓库对象允许根据ID注册顾客，并且对于一个ID只会创建一个Customer对象。有了仓库对象，我就可以修改Order对象的构造函数来使用它。

在使用本重构手法时，可能仓库对象已经存在了，那么就可以直接使用它。

下一步是要弄清楚，Order的构造函数如何获得正确的Customer对象。在这个例子里，这一步很简单，因为输入数据流中已经包含了顾客的ID。

class Order...

```
constructor(data) {
  this._number = data.number;
  this._customer = registerCustomer(data.customer);
  // load other data
}
get customer() {return this._customer;}
```

现在，如果我在一条订单中修改了顾客信息，就会同步反映在该顾客拥有的所有订单中。

在这个例子里，我在第一个引用该顾客信息的Order对象中新建了Customer对象。另一个常见的做法是：首先获取一份包含所有Customer对象的列表，将其填入仓库对象，然后在读取Order对象时关联到对应的Customer对象。如果这样做，那

么Order对象包含的顾客ID必须指向一个仓库中已有的Customer对象，否则就表示程序中有错误。

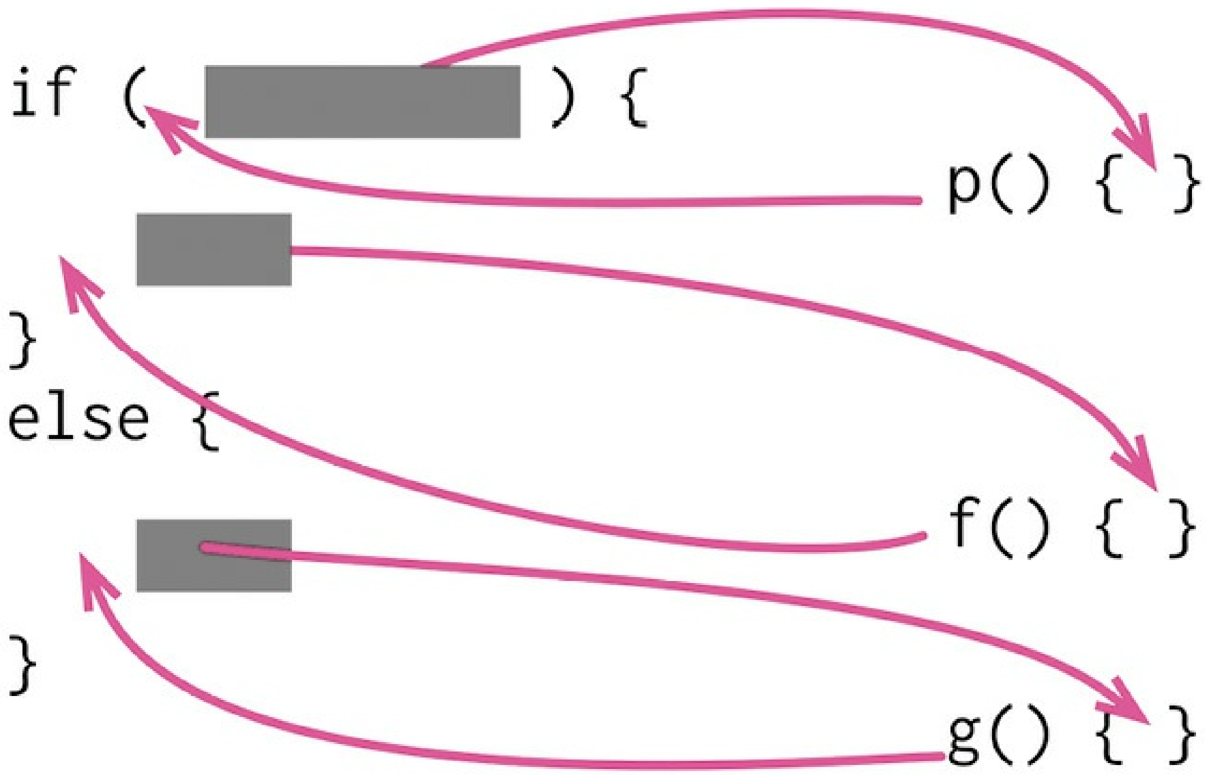
上面的代码还有一个问题：构造函数与一个全局的仓库对象耦合。全局对象必须小心对待：它们就像强力的药物，少用一点儿大有益处，用过量就是毒药。如果想解决这个问题，可以将仓库对象作为参数传递给构造函数。

第10章　简化条件逻辑

程序的大部分威力来自条件逻辑，但很不幸，程序的复杂度也大多来自条件逻辑。我经常借助重构把条件逻辑变得更容易理解。我常用分解条件表达式（260）处理复杂的条件表达式，用合并条件表达式（263）厘清逻辑组合。我会用以卫语句取代嵌套条件表达式（266）清晰表达“在主要处理逻辑之前先做检查”的意图。如果我发现一处`switch`逻辑处理了几种情况，可以考虑拿出以多态取代条件表达式（272）重构手法。

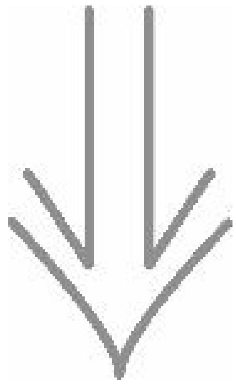
很多条件逻辑是用于处理特殊情况的，例如处理`null`值。如果对某种特殊情况的处理逻辑大多相同，那么可以用引入特例（289）（常被称作引入空对象）消除重复代码。另外，虽然我很喜欢去除条件逻辑，但如果我想明确地表述（以及检查）程序的状态，引入断言（302）是一个不错的补充。

10.1　分解条件表达式（Decompose Conditional）




```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.
  charge = quantity * plan.summerRate;
else

charge = quantity * plan.regularRate + plan.regularServiceCha
```



```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();
```

动机

程序之中，复杂的条件逻辑是最常导致复杂度上升的地点之一。我必须编写代码来检查不同的条件分支，根据不同的条件做不同的事，然后，我很快就会得到一个相当长的函数。大型函数本身就会使代码的可读性下降，而条件逻辑则会使代码更难阅读。在带有复杂条件逻辑的函数中，代码（包括检查条件分支的代码和真正实现功能的代码）会告诉我发生的事，但常常让我弄不清楚为什么会发生这样的事，这就说明代码的可读性的确大大降低了。

和任何大块头代码一样，我可以将它分解为多个独立的函数，根据每个小块代码的用途，为分解而得的新函数命名，并将原函数中对应的代码改为调用新函数，从而更清楚地表达自己的意图。对于条件逻辑，将每个分支条件分解成新函数还可以带来更多好处：可以突出条件逻辑，更清楚地表明每个分支的作用，并且突出每个分支的原因。

本重构手法其实只是提炼函数（106）的一个应用场景。但我要特别强调这个场景，因为我发现它经常会带来很大的价值。

做法

- 对条件判断和每个条件分支分别运用提炼函数（106）手法。

范例

假设我要计算购买某样商品的总价（总价=数量×单价），而这个商品在冬季和夏季的单价是不同的：

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.
  charge = quantity * plan.summerRate;
else

charge = quantity * plan.regularRate + plan.regularServiceCha
```

我把条件判断提炼到一个独立的函数中：

```
if (summer())
  charge = quantity * plan.summerRate;
else

charge = quantity * plan.regularRate + plan.regularServiceCha

function summer() {

return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(pl
}
```

然后提炼条件判断为真的分支：

```
if (summer())
  charge = summerCharge();
else

charge = quantity * plan.regularRate + plan.regularServiceCha

function summer() {

return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(pl
}
function summerCharge() {
  return quantity * plan.summerRate;
}
```

最后提炼条件判断为假的分支：

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();

function summer() {

return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(pl
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {

return quantity * plan.regularRate + plan.regularServiceCharg
}
```

提炼完成后，我喜欢用三元运算符重新安排条件语句。

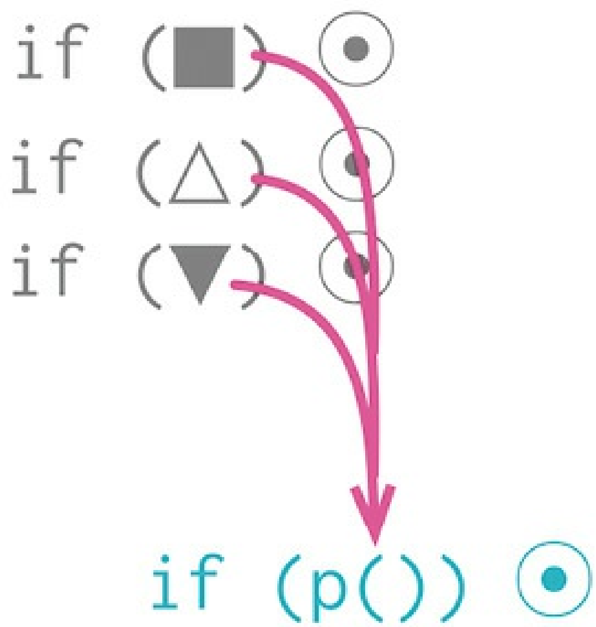
```
charge = summer() ? summerCharge() : regularCharge();

function summer() {

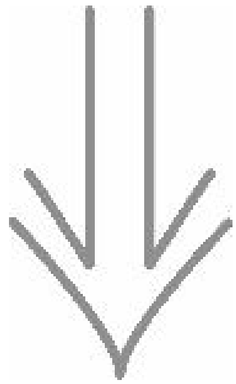
return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(pl
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {

return quantity * plan.regularRate + plan.regularServiceCharg
}
```

10.2 合并条件表达式（Consolidate Conditional Expression）




```
if (anEmployee.seniority < 2) return 0;
if (anEmployee.monthsDisabled > 12) return 0;
if (anEmployee.isPartTime) return 0;
```



```
if (isNotEligibleForDisability()) return 0;

function isNotEligibleForDisability() {
  return ((anEmployee.seniority < 2)
          || (anEmployee.monthsDisabled > 12)
          || (anEmployee.isPartTime));
}
```

动机

有时我会发现这样一串条件检查：检查条件各不相同，最终行为却一致。如果发现这种情况，就应该使用“逻辑或”和“逻辑与”将它们合并为一个条件表达式。

之所以要合并条件代码，有两个重要原因。首先，合并后的条件代码会表述“实际上只有一次条件检查，只不过有多个并列条件需要检查而已”，从而使这一次检查的用意更清晰。当然，合并前和

合并后的代码有着相同的效果，但原先代码传达出的信息却是“这里有一些各自独立的条件测试，它们只是恰好同时发生”。其次，这项重构往往可以为使用提炼函数（106）做好准备。将检查条件提炼成一个独立的函数对于厘清代码意义非常有用，因为它把描述“做什么”的语句换成了“为什么这样做”。

条件语句的合并理由也同时指出了不要合并的理由：如果我认为这些检查的确彼此独立，的确不应该被视为同一次检查，我就不会使用本项重构。

做法

- 确定这些条件表达式都没有副作用。

  如果某个条件表达式有副作用，可以先用将查询函数和修改函数分离（306）处理。

- 使用适当的逻辑运算符，将两个相关条件表达式合并为一个。

顺序执行的条件表达式用逻辑或来合并，嵌套的if语句用逻辑与来合并。

- 测试。
- 重复前面的合并过程，直到所有相关的条件表达式都合并到一起。
- 可以考虑对合并后的条件表达式实施提炼函数（106）。

范例

在走读代码的过程中，我看到了下面的代码片段：

```
function disabilityAmount(anEmployee) {
  if (anEmployee.seniority < 2) return 0;
  if (anEmployee.monthsDisabled > 12) return 0;
  if (anEmployee.isPartTime) return 0;
  // compute the disability amount
```

这里有一连串的条件检查，都指向同样的结果。既然结果是相同的，就应该把这些条件检查合并成一条表达式。对于这样顺序执行的条件检查，可以用逻辑或运算符来合并。

```
function disabilityAmount(anEmployee) {
  if ((anEmployee.seniority < 2)
      || (anEmployee.monthsDisabled > 12)) return 0;
  if (anEmployee.isPartTime) return 0;
  // compute the disability amount
```

测试，然后把下一个条件检查也合并进来：

```
function disabilityAmount(anEmployee) {
  if ((anEmployee.seniority < 2)
      || (anEmployee.monthsDisabled > 12)
      || (anEmployee.isPartTime)) return 0;
  // compute the disability amount
```

合并完成后，再对这句条件表达式使用提炼函数（106）。

```
function disabilityAmount(anEmployee) {
  if (isNotEligableForDisability()) return 0;
  // compute the disability amount

function isNotEligableForDisability() {
  return ((anEmployee.seniority < 2)
         || (anEmployee.monthsDisabled > 12)
         || (anEmployee.isPartTime));
}
```

范例：使用逻辑与

上面的例子展示了用逻辑或合并条件表达式的做法。不过，我有可能遇到需要逻辑与的情况。例如，嵌套if语句的情况：

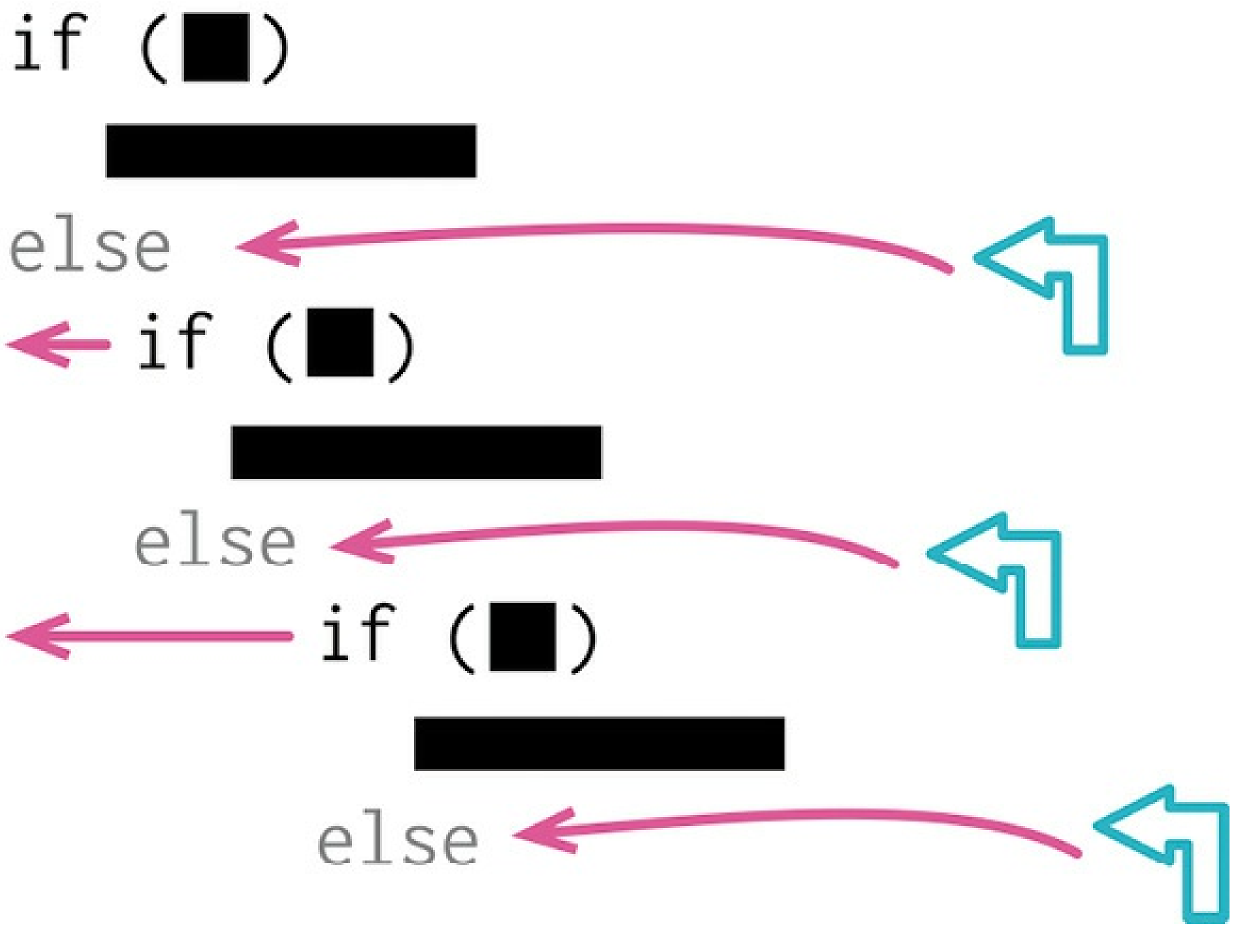
```
if (anEmployee.onVacation)
  if (anEmployee.seniority > 10)
    return 1;
return 0.5;
```

可以用逻辑与运算符将其合并。

```
if ((anEmployee.onVacation)
    && (anEmployee.seniority > 10)) return 1;
return 0.5;
```

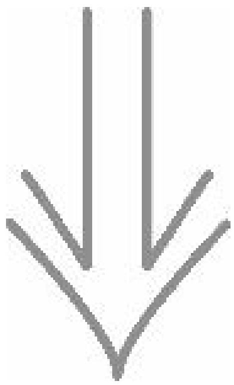
如果原来的条件逻辑混杂了这两种情况，我也会根据需要组合使用逻辑与和逻辑或运算符。在这种时候，代码很可能变得混乱，所以我会频繁使用提炼函数（106），把代码变得可读。

10.3 以卫语句取代嵌套条件表达式（Replace Nested Conditional with Guard Clauses）




```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
  else {
    if (isSeparated)
```

```
      result = separatedAmount();
    else {
      if (isRetired)
        result = retiredAmount();
      else
        result = normalPayAmount();
    }
  }
  return result;
}
```



```
function getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```

动机

根据我的经验，条件表达式通常有两种风格。第一种风格是：两个条件分支都属于正常行为。第

二种风格则是：只有一个条件分支是正常行为，另一个分支则是异常的情况。

这两类条件表达式有不同的用途，这一点应该通过代码表现出来。如果两条分支都是正常行为，就应该使用形如`if...else...`的条件表达式；如果某个条件极其罕见，就应该单独检查该条件，并在该条件为真时立刻从函数中返回。这样的单独检查常常被称为“卫语句”（guard clauses）。

以卫语句取代嵌套条件表达式的精髓就是：给某一条分支以特别的重视。如果使用`if-then-else`结构，你对`if`分支和`else`分支的重视是同等的。这样的代码结构传递给阅读者的消息就是：各个分支有同样的重要性。卫语句就不同了，它告诉阅读者：“这种情况不是本函数的核心逻辑所关心的，如果它真发生了，请做一些必要的整理工作，然后退出。”

“每个函数只能有一个入口和一个出口”的观念，根深蒂固于某些程序员的脑海里。我发现，当我处理他们编写的代码时，经常需要使用以卫语句取代嵌套条件表达式。现今的编程语言都会强制保证每个函数只有一个入口，至于“单一出口”规则，其实不是那么有用。在我看来，保持代码清晰才是最关键的：如果单一出口能使这个函数更清楚易读，那么就使用单一出口；否则就不必这么做。

做法

- 选中最外层需要被替换的条件逻辑，将其替换为卫语句。
- 测试。
- 有需要的话，重复上述步骤。
- 如果所有卫语句都引发同样的结果，可以使用合并条件表达式（263）合并之。

范例

下面的代码用于计算要支付给员工（employee）的工资。只有还在公司上班的员工才需要支付工资，所以这个函数需要检查两种“员工已经不在公司上班”的情况。

```
function payAmount(employee) {
  let result;
  if(employee.isSeparated) {
    result = {amount: 0, reasonCode:"SEP"};
  }
  else {
    if (employee.isRetired) {
      result = {amount: 0, reasonCode: "RET"};
    }
    else {
```

```
      // logic to compute amount
      lorem.ipsum(dolor.sitAmet);[1]
      consectetur(adipiscing).elit();

sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna
      ut.enim.ad(minim.veniam);
      result = someFinalComputation();
    }
  }
  return result;
}
```

嵌套的条件逻辑让我们看不清代码真实的含义。只有当前两个条件表达式都不为真的时候，这段代码才真正开始它的主要工作。所以，卫语句能让代码更清晰地阐述自己的意图。

一如既往地，我喜欢小步前进，所以我先处理最顶上的条件逻辑。

```
function payAmount(employee) {
  let result;

if (employee.isSeparated) return {amount: 0, reasonCode: "SEP
  if (employee.isRetired) {
    result = {amount: 0, reasonCode: "RET"};
  }
  else {
    // logic to compute amount
    lorem.ipsum(dolor.sitAmet);
    consectetur(adipiscing).elit();

sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna
    ut.enim.ad(minim.veniam);
    result = someFinalComputation();
  }
  return result;
}
```

做完这步修改，我执行测试，然后继续下一步。

```
function payAmount(employee) {
  let result;

if (employee.isSeparated) return {amount: 0, reasonCode: "SEP

if (employee.isRetired)   return {amount: 0, reasonCode: "RET
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();

sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna
  ut.enim.ad(minim.veniam);
  result = someFinalComputation();
  return result;
}
```

此时，result变量已经没有用处了，所以我把它删掉：

```
function payAmount(employee) {
  let result;

if (employee.isSeparated) return {amount: 0, reasonCode: "SEP

if (employee.isRetired)   return {amount: 0, reasonCode: "RET
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();

sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna
  ut.enim.ad(minim.veniam);
  return someFinalComputation();
}
```

能减少一个可变变量总是好的。

范例：将条件反转

审阅本书第1版的初稿时，Joshua Kerievsky指出：我们常常可以将条件表达式反转，从而实现以卫语句取代嵌套条件表达式。为了拯救我可怜的想象力，他还好心帮我想了一个例子：

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital > 0) {

if (anInstrument.interestRate > 0 && anInstrument.duration >

result = (anInstrument.income / anInstrument.duration) * anIn
    }
  }
  return result;
}
```

同样地，我逐一进行替换。不过这次在插入卫语句时，我需要将相应的条件反转过来：

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;

if (anInstrument.interestRate > 0 && anInstrument.duration >

result = (anInstrument.income / anInstrument.duration) * anIn
```

```
  }
  return result;
}
```

下一个条件稍微复杂一点，所以我分两步进行反转。首先加入一个逻辑非操作：

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if                                                    (!
(anInstrument.interestRate > 0 && anInstrument.duration > 0))

result = (anInstrument.income / anInstrument.duration) * anIn
  return result;
}
```

但是在这样的条件表达式中留下一个逻辑非，会把我的脑袋拧成一团乱麻，所以我把它简化成下面这样：

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;

if (anInstrument.interestRate <= 0 || anInstrument.duration <

result = (anInstrument.income / anInstrument.duration) * anIn
  return result;
}
```

这两行逻辑语句引发的结果一样，所以我可以

用合并条件表达式（263）将其合并。

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (   anInstrument.capital      <= 0
      || anInstrument.interestRate <= 0
      || anInstrument.duration     <= 0) return result;

result = (anInstrument.income / anInstrument.duration) * anIn
  return result;
}
```

此时result变量做了两件事：一开始我把它设为0，代表卫语句被触发时的返回值；然后又用最终计算的结果给它赋值。我可以彻底移除这个变量，避免用一个变量承担两重责任，而且又减少了一个可变变量。

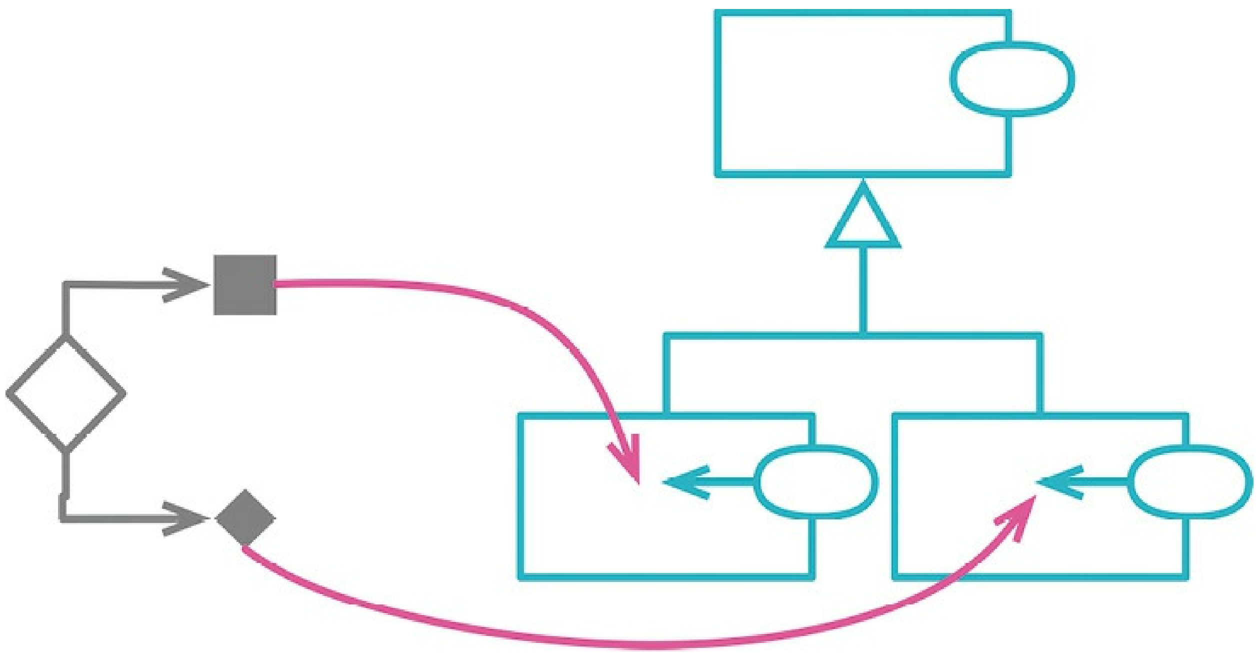
```
function adjustedCapital(anInstrument) {
  if (   anInstrument.capital      <= 0
      || anInstrument.interestRate <= 0
      || anInstrument.duration    <= 0) return 0;

return (anInstrument.income / anInstrument.duration) * anInst
}
```

---

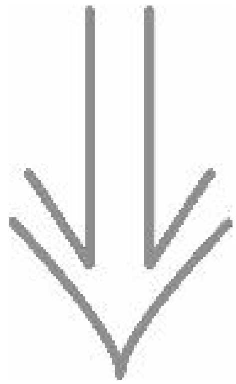
1 “lorem.ipsum……”是一篇常见于排版设计领域的文章，其内容为不具可读性的字符组合，目的是使阅读者只专注于观察段落的字型和版型。——译者注

10.4　以多态取代条件表达式（Replace Conditional with Polymorphism）



```
switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':

return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
```



```
class EuropeanSwallow {
  get plumage() {
    return "average";
  }
class AfricanSwallow {
  get plumage() {

 return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
class NorwegianBlueParrot {
  get plumage() {
     return (this.voltage > 100) ? "scorched" : "beautiful";
}
```

动机

复杂的条件逻辑是编程中最难理解的东西之一，因此我一直在寻求给条件逻辑添加结构。很多时候，我发现可以将条件逻辑拆分到不同的场景（或者叫高阶用例），从而拆解复杂的条件逻辑。这种拆分有时用条件逻辑本身的结构就足以表达，但使用类和多态能把逻辑的拆分表述得更清晰。

一个常见的场景是：我可以构造一组类型，每个类型处理各自的一种条件逻辑。例如，我会注意到，图书、音乐、食品的处理方式不同，这是因为它们分属不同类型的商品。最明显的征兆就是有好几个函数都有基于类型代码的switch语句。若果真如此，我就可以针对switch语句中的每种分支逻辑创建一个类，用多态来承载各个类型特有的行为，从而去除重复的分支逻辑。

另一种情况是：有一个基础逻辑，在其上又有一些变体。基础逻辑可能是最常用的，也可能是最简单的。我可以把基础逻辑放进超类，这样我可以首先理解这部分逻辑，暂时不管各种变体，然后我可以把每种变体逻辑单独放进一个子类，其中的代码着重强调与基础逻辑的差异。

多态是面向对象编程的关键特性之一。跟其他一切有用的特性一样，它也很容易被滥用。我曾经遇到有人争论说所有条件逻辑都应该用多态取代。我不赞同这种观点。我的大部分条件逻辑只用到了基本的条件语句——if/else和switch/case，并不需要劳师动众地引入多态。但如果发现如前所述的复杂条件逻辑，多态是改善这种情况的有力工具。

做法

- 如果现有的类尚不具备多态行为，就用工厂函数创建之，令工厂函数返回恰当的对象实例。
- 在调用方代码中使用工厂函数获得对象实例。
- 将带有条件逻辑的函数移到超类中。

如果条件逻辑还未提炼至独立的函数，首先对其使用提炼函数（106）。

- 任选一个子类，在其中建立一个函数，使之覆写超类中容纳条件表达式的那个函数。将与该子类相关的条件表达式分支复制到新函数中，并对它进行适当调整。
- 重复上述过程，处理其他条件分支。
- 在超类函数中保留默认情况的逻辑。或者，如果超类应该是抽象的，就把该函数声明为`abstract`，或在其中直接抛出异常，表明计算责任都在子类中。

范例

我的朋友有一群鸟儿，他想知道这些鸟飞得有多快，以及它们的羽毛是什么样的。所以我们写了一小段程序来判断这些信息。

```
function plumages(birds) {
  return new Map(birds.map(b => [b.name, plumage(b)]));
}
function speeds(birds) {

return new Map(birds.map(b => [b.name, airSpeedVelocity(b)]))
}

function plumage(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':

return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}

function airSpeedVelocity(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return 35;
  case 'AfricanSwallow':
    return 40 - 2 * bird.numberOfCoconuts;
  case 'NorwegianBlueParrot':
    return (bird.isNailed) ? 0 : 10 + bird.voltage / 10;
  default:
    return null;
  }
}
```

有两个不同的操作，其行为都随着“鸟的类

型”发生变化，因此可以创建出对应的类，用多态来处理各类型特有的行为。

我先对airSpeedVelocity和plumage两个函数使用函数组合成类（144）。

```
function plumage(bird) {
  return new Bird(bird).plumage;
}

function airSpeedVelocity(bird) {
  return new Bird(bird).airSpeedVelocity;
}
class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return "average";
    case 'AfricanSwallow':

return (this.numberOfCoconuts > 2) ? "tired" : "average";
    case 'NorwegianBlueParrot':

return (this.voltage > 100) ? "scorched" : "beautiful";
    default:
      return "unknown";
    }
  }
  get airSpeedVelocity() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return 35;
    case 'AfricanSwallow':
      return 40 - 2 * this.numberOfCoconuts;
    case 'NorwegianBlueParrot':
      return (this.isNailed) ? 0 : 10 + this.voltage / 10;
    default:
      return null;
    }
  }
```

```
}
```

然后针对每种鸟创建一个子类，用一个工厂函数来实例化合适的子类对象。

```
function plumage(bird) {
  return createBird(bird).plumage;
}

function airSpeedVelocity(bird) {
  return createBird(bird).airSpeedVelocity;
}

  function createBird(bird) {
    switch (bird.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(bird);
    case 'AfricanSwallow':
      return new AfricanSwallow(bird);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(bird);
    default:
      return new Bird(bird);
    }
  }
  class EuropeanSwallow extends Bird {
  }

  class AfricanSwallow extends Bird {
  }

  class NorwegianBlueParrot extends Bird {
  }
```

现在我已经有了需要的类结构，可以处理两个条件逻辑了。先从plumage函数开始，我从switch语句中选一个分支，在适当的子类中覆写这个逻

辑。

class EuropeanSwallow...

```
get plumage() {
  return "average";
}
```

class Bird...

```
get plumage() {
  switch (this.type) {
  case 'EuropeanSwallow':
    throw "oops";
  case 'AfricanSwallow':

return (this.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (this.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}
```

在超类中，我把对应的逻辑分支改为抛出异常，因为我总是偏执地担心出错。

此时我就可以编译并测试。如果一切顺利的话，我可以接着处理下一个分支。

class AfricanSwallow...

```
get plumage() {
   return (this.numberOfCoconuts > 2) ? "tired" : "average";
}
```

然后是挪威蓝鹦鹉（Norwegian Blue）的分支。

class NorwegianBlueParrot...

```
get plumage() {
   return (this.voltage >100) ? "scorched" : "beautiful";
}
```

超类函数保留下来处理默认情况。

class Bird...

```
get plumage() {
  return "unknown";
}
```

`airSpeedVelocity`也如法炮制。完成以后，代

码大致如下（我还对顶层的airSpeedVelocity和plumage函数做了内联处理）：

```
function plumages(birds) {
  return new Map(birds
                .map(b => createBird(b))
                .map(bird => [bird.name, bird.plumage]));
}
function speeds(birds) {
  return new Map(birds
                .map(b => createBird(b))

 .map(bird => [bird.name, bird.airSpeedVelocity]));
}

function createBird(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return new EuropeanSwallow(bird);
  case 'AfricanSwallow':
    return new AfricanSwallow(bird);
  case 'NorwegianBlueParrot':
    return new NorwegianBlueParrot(bird);
  default:
    return new Bird(bird);
  }
}

class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    return "unknown";
  }
  get airSpeedVelocity() {
    return null;
  }
}
class EuropeanSwallow extends Bird {
  get plumage() {
    return "average";
  }
  get airSpeedVelocity() {
    return 35;
```

```
    }
  }
  class AfricanSwallow extends Bird {
    get plumage() {
      return (this.numberOfCoconuts > 2) ? "tired" : "average";
    }
    get airSpeedVelocity() {
      return 40 - 2 * this.numberOfCoconuts;
    }
  }
  class NorwegianBlueParrot extends Bird {
    get plumage() {
      return (this.voltage > 100) ? "scorched" : "beautiful";
    }
    get airSpeedVelocity() {
      return (this.isNailed) ? 0 : 10 + this.voltage / 10;
    }
  }
```

看着最终的代码，可以看出Bird超类并不是必需的。在JavaScript中，多态不一定需要类型层级，只要对象实现了适当的函数就行。但在这个例子中，我愿意保留这个不必要的超类，因为它能帮助阐释各个子类与问题域之间的关系。

范例：用多态处理变体逻辑

在前面的例子中，“鸟”的类型体系是一个清晰的泛化体系：超类是抽象的“鸟”，子类是各种具体的鸟。这是教科书（包括我写的书）中经常讨论的继承和多态，但并不是实践中使用继承的唯一方

式。实际上，这种方式很可能不是最常用或最好的方式。另一种使用继承的情况是：我想表达某个对象与另一个对象大体类似，但又有一些不同之处。

下面有一个这样的例子：有一家评级机构，要对远洋航船的航行进行投资评级。这家评级机构会给出“A”或者“B”两种评级，取决于多种风险和盈利潜力的因素。在评估风险时，既要考虑航程本身的特征，也要考虑船长过往航行的历史。

```
function rating(voyage, history) {
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}
function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if                    (["china",                    "east-
indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}
function captainHistoryRisk(voyage, history) {
  let result = 1;
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
  if (voyage.zone === "china" && hasChina(history)) result -
= 2;
  return Math.max(result, 0);
}
function hasChina(history) {
  return history.some(v => "china" === v.zone);
}
function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
```

```
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

voyageRisk和captainHistoryRisk两个函数负责打出风险分数，voyageProfitFactor负责打出盈利潜力分数，rating函数将3个分数组合到一起，给出一次航行的综合评级。

调用方的代码大概是这样：

```
const voyage = {zone: "west-indies", length: 10};
const history = [
  {zone: "east-indies", profit: 5},
  {zone: "west-indies", profit: 15},
  {zone: "china",      profit: -2},
  {zone: "west-africa", profit: 7},
];

const myRating = rating(voyage, history);
```

代码中有两处同样的条件逻辑，都在询问“是否有到中国的航程”以及“船长是否曾去过中国”。

```
function rating(voyage, history) {
```

```
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}
  function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if                    (["china",                        "east-
indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}
function captainHistoryRisk(voyage, history) {
  let result = 1;
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
  if (voyage.zone === "china" && hasChina(history)) result -
= 2;
  return Math.max(result, 0);
}
function hasChina(history) {
  return history.some(v => "china" === v.zone);
}
function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

我会用继承和多态将处理“中国因素”的逻辑从基础逻辑中分离出来。如果还要引入更多的特殊逻

辑，这个重构就很有用——这些重复的“中国因素”会混淆视听，让基础逻辑难以理解。

起初代码里只有一堆函数，如果要引入多态的话，我需要先建立一个类结构，因此我首先使用函数组合成类（144）。这一步重构的结果如下所示：

```
function rating(voyage, history) {
  return new Rating(voyage, history).value;
}

class Rating {
  constructor(voyage, history) {
    this.voyage = voyage;
    this.history = history;
  }
  get value() {
    const vpf = this.voyageProfitFactor;
    const vr = this.voyageRisk;
    const chr = this.captainHistoryRisk;
    if (vpf * 3 > (vr + chr * 2)) return "A";
    else return "B";
  }
  get voyageRisk() {
    let result = 1;
    if (this.voyage.length > 4) result += 2;

if (this.voyage.length > 8) result += this.voyage.length - 8;
    if                    (["china",                        "east-
indies"].includes(this.voyage.zone)) result += 4;
    return Math.max(result, 0);
  }
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;

if (this.voyage.zone === "china" && this.hasChinaHistory) res
= 2;
    return Math.max(result, 0);
```

```
  }
  get voyageProfitFactor() {
    let result = 2;

    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;

if (this.voyage.zone === "china" && this.hasChinaHistory) {
      result += 3;
      if (this.history.length > 10) result += 1;
      if (this.voyage.length > 12) result += 1;
      if (this.voyage.length > 18) result -= 1;
    }
    else {
      if (this.history.length > 8) result += 1;
      if (this.voyage.length > 14) result -= 1;
    }
    return result;
  }
  get hasChinaHistory() {
    return this.history.some(v => "china" === v.zone);
  }
}
```

于是我就有了一个类，用来安放基础逻辑。现在我需要另建一个空的子类，用来安放与超类不同的行为。

```
class ExperiencedChinaRating extends Rating {
}
```

然后，建立一个工厂函数，用于在需要时返回变体类。

```
function createRating(voyage, history) {
```

```
if (voyage.zone === "china" && history.some(v => "china" ===
    return new ExperiencedChinaRating(voyage, history);
  else return new Rating(voyage, history);
}
```

我需要修改所有调用方代码，让它们使用该工厂函数，而不要直接调用构造函数。还好现在调用构造函数的只有rating函数一处。

```
function rating(voyage, history) {
  return createRating(voyage, history).value;
}
```

有两处行为需要移入子类中。我先处理captainHistoryRisk中的逻辑。

class Rating...

```
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;

if (this.voyage.zone === "china" && this.hasChinaHistory) res
= 2;
  return Math.max(result, 0);
}
```

在子类中覆写这个函数。

class ExperiencedChinaRating

```
get captainHistoryRisk() {
  const result = super.captainHistoryRisk - 2;
  return Math.max(result, 0);
}
```

class Rating...

```
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;
  if (this.voyage.zone === "china" && this.hasChinaHistory) r
= 2;
  return Math.max(result, 0);
}
```

分离voyageProfitFactor函数中的变体行为要更麻烦一些。我不能直接从超类中删掉变体行为，因为在超类中还有另一条执行路径。我又不想把整个超类中的函数复制到子类中。

class Rating...

```
get voyageProfitFactor() {
  let result = 2;
```

```
  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;

if (this.voyage.zone === "china" && this.hasChinaHistory) {
    result += 3;
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
  }
  else {
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
  }
  return result;
}
```

所以我先用提炼函数（106）将整个条件逻辑块提炼出来。

class Rating...

```
get voyageProfitFactor() {
  let result = 2;

  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.voyageAndHistoryLengthFactor;
  return result;
}
get voyageAndHistoryLengthFactor() {
  let result = 0;

if (this.voyage.zone === "china" && this.hasChinaHistory) {
    result += 3;
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
  }
  else {
```

```
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
  }
  return result;
}
```

函数名中出现“And”字样是一个很不好的味道，不过我会暂时容忍它，先聚焦子类化操作。

class Rating...

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  if (this.history.length > 8) result += 1;
  if (this.voyage.length > 14) result -= 1;
  return result;
}
```

class ExperiencedChinaRating...

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  if (this.history.length > 10) result += 1;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

严格说来，重构到这儿就结束了——我已经把变体行为分离到了子类中，超类的逻辑理解和维护起来更简单了，只有在进入子类代码时我才需要操心变体逻辑。子类的代码表述了它与超类的差异。

但我觉得至少应该谈谈如何处理这个丑陋的新函数。引入一个函数以便子类覆写，这在处理这种“基础和变体”的继承关系时是常见操作。但这样一个难看的函数只会妨碍——而不是帮助——别人理解其中的逻辑。

函数名中的“And”字样说明其中包含了两件事，所以我觉得应该将它们分开。我会用提炼函数（106）把“历史航行数”（history length）的相关逻辑提炼出来。这一步提炼在超类和子类中都要发生，我首先从超类开始。

class Rating...

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += this.historyLengthFactor;
  if (this.voyage.length > 14) result -= 1;
  return result;
}
get historyLengthFactor() {
  return (this.history.length > 8) ? 1 : 0;
}
```

然后在子类中也如法炮制。

class ExperiencedChinaRating...

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  result += this.historyLengthFactor;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
get historyLengthFactor() {
  return (this.history.length > 10) ? 1 : 0;
}
```

然后在超类中使用搬移语句到调用者（217）。

class Rating...

```
get voyageProfitFactor() {
  let result = 2;
  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.historyLengthFactor;
  result += this.voyageAndHistoryLengthFactor;
  return result;
}

get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += this.historyLengthFactor;
  if (this.voyage.length > 14) result -= 1;
```

```
  return result;
}
```

class ExperiencedChinaRating...

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  result += this.historyLengthFactor;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

再用函数改名（124）改掉这个难听的名字。

class Rating...

```
get voyageProfitFactor() {
  let result = 2;
  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.historyLengthFactor;
  result += this.voyageLengthFactor;
  return result;
}

get voyageLengthFactor() {
  return (this.voyage.length > 14) ? - 1: 0;
}
```

改为三元表达式，以简化voyageLengthFactor函数。

class ExperiencedChinaRating...

```
get voyageLengthFactor() {
  let result = 0;
  result += 3;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

最后一件事：在“航程数”（voyage length）因素上加上3分，我认为这个逻辑不合理，应该把这3分加在最终的结果上。

class ExperiencedChinaRating...

```
get voyageProfitFactor() {
  return super.voyageProfitFactor + 3;
}

get voyageLengthFactor() {
  let result = 0;
  result += 3;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

重构结束，我得到了如下代码。首先，我有一个基本的Rating类，其中不考虑与“中国经验”相关的复杂性：

```
class Rating {
  constructor(voyage, history) {
    this.voyage = voyage;
    this.history = history;
  }
  get value() {
    const vpf = this.voyageProfitFactor;
    const vr = this.voyageRisk;
    const chr = this.captainHistoryRisk;
    if (vpf * 3 > (vr + chr * 2)) return "A";
    else return "B";
  }
  get voyageRisk() {
    let result = 1;
    if (this.voyage.length > 4) result += 2;

if (this.voyage.length > 8) result += this.voyage.length - 8;
    if                    (["china",                     "east-
indies"].includes(this.voyage.zone)) result += 4;
    return Math.max(result, 0);
  }
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;
    return Math.max(result, 0);
  }
  get voyageProfitFactor() {
    let result = 2;
    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    result += this.historyLengthFactor;
    result += this.voyageLengthFactor;
    return result;
  }
  get voyageLengthFactor() {
    return (this.voyage.length > 14) ? - 1: 0;
  }
  get historyLengthFactor() {
    return (this.history.length > 8) ? 1 : 0;
```

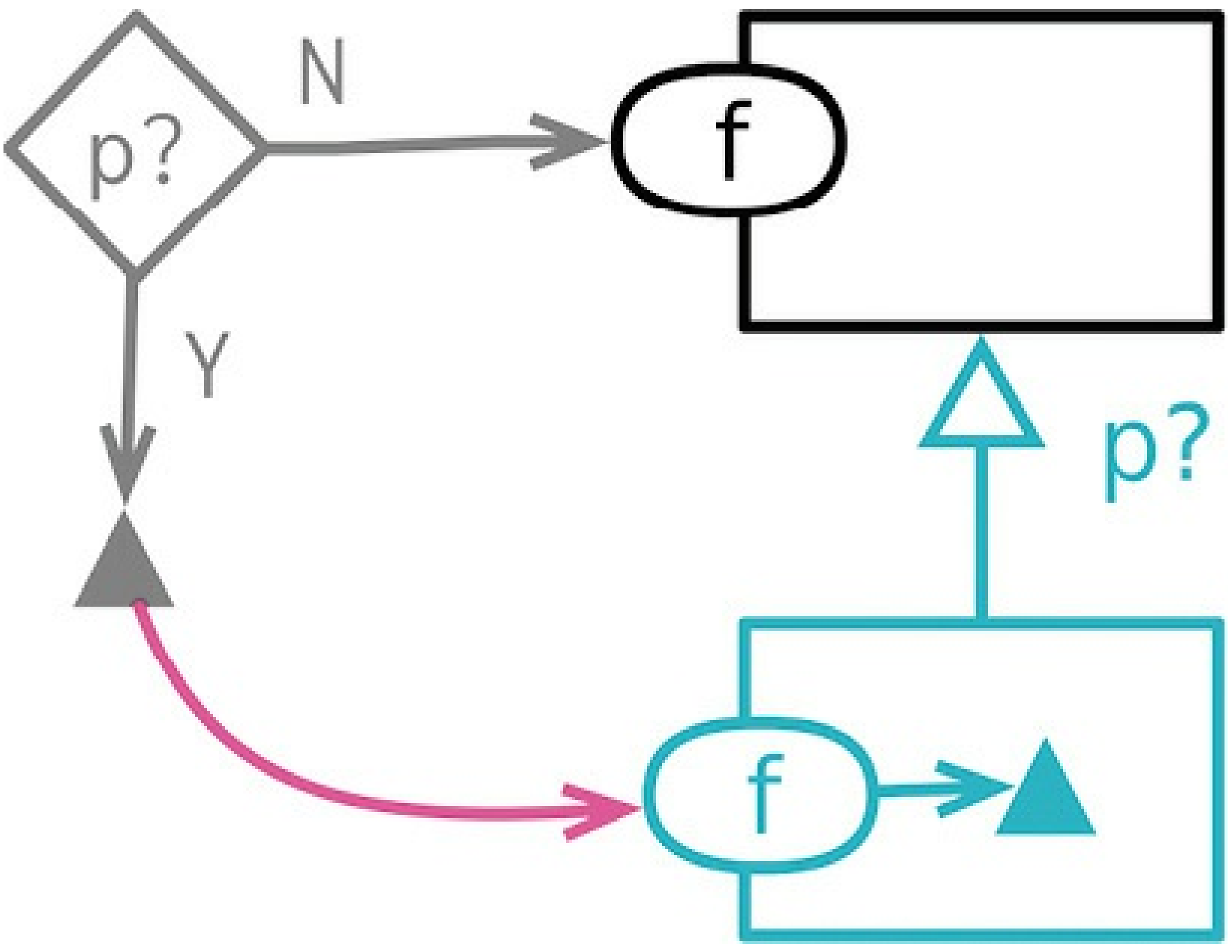
```
  }
}
```

与“中国经验”相关的代码则清晰表述出在基本逻辑之上的一系列变体逻辑：

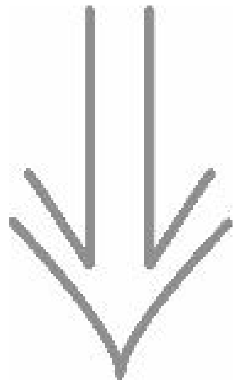
```
class ExperiencedChinaRating extends Rating {
  get captainHistoryRisk() {
    const result = super.captainHistoryRisk - 2;
    return Math.max(result, 0);
  }
  get voyageLengthFactor() {
    let result = 0;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
  get historyLengthFactor() {
    return (this.history.length > 10) ? 1 : 0;
  }
  get voyageProfitFactor() {
    return super.voyageProfitFactor + 3;
  }
}
```

10.5　引入特例（Introduce Special Case）

曾用名：引入Null对象（Introduce Null Object）




```
if (aCustomer === "unknown") customerName = "occupant";
```



```
class UnknownCustomer {
    get name() {return "occupant";}
```

动机

一种常见的重复代码是这种情况：一个数据结构的使用者都在检查某个特殊的值，并且当这个特殊值出现时所做的处理也都相同。如果我发现代码库中有多处以同样方式应对同一个特殊值，我就会想要把这个处理逻辑收拢到一处。

处理这种情况的一个好办法是使用“特例”（Special Case）模式：创建一个特例元素，用以表达对这种特例的共用行为的处理。这样我就可以用一个函数调用取代大部分特例检查逻辑。

特例有几种表现形式。如果我只需要从这个对

象读取数据，可以提供一个字面量对象（literal object），其中所有的值都是预先填充好的。如果除简单的数值之外还需要更多的行为，就需要创建一个特殊对象，其中包含所有共用行为所对应的函数。特例对象可以由一个封装类来返回，也可以通过变换插入一个数据结构。

一个通常需要特例处理的值就是`null`，这也是这个模式常被叫作“Null对象”（Null Object）模式的原因——我喜欢说：Null对象是特例的一种特例。

做法

我们从一个作为容器的数据结构（或者类）开始，其中包含一个属性，该属性就是我们要重构的目标。容器的客户端每次使用这个属性时，都需要将其与某个特例值做比对。我们希望把这个特例值替换为代表这种特例情况的类或数据结构。

- 给重构目标添加检查特例的属性，令其返回`false`。
- 创建一个特例对象，其中只有检查特例的属性，返回`true`。

- 对“与特例值做比对”的代码运用提炼函数（106），确保所有客户端都使用这个新函数，而不再直接做特例值的比对。
- 将新的特例对象引入代码中，可以从函数调用中返回，也可以在变换函数中生成。
- 修改特例比对函数的主体，在其中直接使用检查特例的属性。
- 测试。
- 使用函数组合成类（144）或函数组合成变换（149），把通用的特例处理逻辑都搬移到新建的特例对象中。

特例类对于简单的请求通常会返回固定的值，因此可以将其实现为字面记录（literal record）。

- 对特例比对函数使用内联函数（115），将其内联到仍然需要的地方。

范例

一家提供公共事业服务的公司将自己的服务安

装在各个场所（site）。

class Site...

```
get customer() {return this._customer;}
```

代表“顾客”的Customer类有多个属性，我只考虑其中3个。

class Customer...

```
get name()            {...}
get billingPlan()     {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

大多数情况下，一个场所会对应一个顾客，但有些场所没有与之对应的顾客，可能是因为之前的住户搬走了，而新搬来的住户我还不知道是谁。这种情况下，数据记录中的customer字段会被填充为字符串"unknown"。因为这种情况时有发生，所以Site对象的客户端必须有办法处理“顾客未知”的情况。下面是一些示例代码片段。

客户端1...

```
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

客户端2...

```
const plan = (aCustomer === "unknown") ?
     registry.billingPlans.basic
     : aCustomer.billingPlan;
```

客户端3...

```
if (aCustomer !== "unknown") aCustomer.billingPlan = newPlan;
```

客户端4...

```
const weeksDelinquent = (aCustomer === "unknown") ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

浏览整个代码库，我看到有很多使用Site对象的客户端在处理“顾客未知”的情况，大多数都用了同样的应对方式：用"occupant"（居民）作为顾客名，使用基本的计价套餐，并认为这家顾客没有欠费。到处都在检查这种特例，再加上对特例的处理方式高度一致，这些现象告诉我：是时候使用特例对象（Special Case Object）模式了。

我首先给Customer添加一个函数，用于指示“这个顾客是否未知”。

class Customer...

```
get isUnknown() {return false;}
```

然后我给“未知的顾客”专门创建一个类。

```
class UnknownCustomer {
  get isUnknown() {return true;}
}
```

注意，我没有把UnknownCustomer类声明为Customer的子类。在其他编程语言（尤其是静态类型的编程语言）中，我会需要继承关

系。但JavaScript是一种动态类型语言，按照它的子类化规则，这里不声明继承关系反而更好。

下面就是麻烦之处了。我必须在所有期望得到"unknown"值的地方返回这个新的特例对象，并修改所有检查"unknown"值的地方，令其使用新的isUnknown函数。一般而言，我总是希望细心安排修改过程，使我可以每次做一点小修改，然后马上测试。但如果我修改了Customer类，使其返回UnknownCustomer对象（而非"unknown"字符串），那么就必须同时修改所有客户端，让它们不要检查"unknown"字符串，而是调用isUnknown函数——这两个修改必须一次完成。我感觉这一大步修改就像一大块难吃的食物一样难以下咽。

还好，遇到这种困境时，有一个常用的技巧可以帮忙。如果有一段代码需要在很多地方做修改（例如我们这里的“与特例做比对”的代码），我会先对其使用提炼函数（106）。

```
function isUnknown(arg) {
  if (!((arg instanceof Customer) || (arg === "unknown")))
    throw new Error(`investigate bad value: <${arg}>`);
  return (arg === "unknown");
}
```

我会放一个陷阱，捕捉意料之外的值。如果在重构过程中我犯了错误，引入了奇怪的行为，这个陷阱会帮我发现。

现在，凡是检查未知顾客的地方，都可以改用这个函数了。我可以逐一修改这些地方，每次修改之后都可以执行测试。

客户端1...

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

没用多久，就全部修改完了。

客户端2...

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

客户端3...

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

客户端4...

```
const weeksDelinquent = isUnknown(aCustomer) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

将所有调用处都改为使用isUnknown函数之后，就可以修改Site类，令其在顾客未知时返回UnknownCustomer对象。

class Site...

```
get customer() {
  return (this._customer === "unknown") ? new UnknownCustomer
}
```

然后修改isUnknown函数的判断逻辑。做完这步修改之后我可以做一次全文搜索，应该没有任何地方使用"unknown"字符串了。

客户端1...

```
function isUnknown(arg) {
  if                                                          (!
(arg instanceof Customer || arg instanceof UnknownCustomer))
    throw new Error(`investigate bad value: <${arg}>`);
  return arg.isUnknown;
}
```

测试，以确保一切运转如常。

现在，有趣的部分开始了。我可以逐一查看客户端检查特例的代码，看它们处理特例的逻辑，并考虑是否能用函数组合成类（144）将其替换为一个共同的、符合预期的值。此刻，有多处客户端代码用字符串"occupant"来作为未知顾客的名字，就像下面这样。

客户端1...

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

我可以在UnknownCustomer类中添加一个合适的函数。

class UnknownCustomer...

```
get name() {return "occupant";}
```

然后我就可以去掉所有条件代码。

客户端**1...**

```
const customerName = aCustomer.name;
```

测试通过之后，我可能会用内联变量（123）把`customerName`变量也消除掉。

接下来处理代表“计价套餐”的`billingPlan`属性。

客户端**2...**

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

客户端**3...**

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

对于读取该属性的行为，我的处理方法跟前面处理name属性一样——找到通用的应对方式，并在UnknownCustomer中使用之。至于对该属性的写操作，当前的代码没有对未知顾客调用过设值函数，所以在特例对象中，我会保留设值函数，但其中什么都不做。

class UnknownCustomer...

```
get billingPlan()  {return registry.billingPlans.basic;}
set billingPlan(arg) { /* ignore */ }
```

读取的例子...

```
const plan = aCustomer.billingPlan;
```

更新的例子...

```
aCustomer.billingPlan = newPlan;
```

特例对象是值对象，因此应该始终是不可变

的，即便它们替代的原对象本身是可变的。

最后一个例子则更麻烦一些，因为特例对象需要返回另一个对象，后者又有其自己的属性。

客户端...

```
const weeksDelinquent = isUnknown(aCustomer) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

一般的原则是：如果特例对象需要返回关联对象，被返回的通常也是特例对象。所以，我需要创建一个代表“空支付记录”的特例类NullPaymentHistory。

class UnknownCustomer...

```
get paymentHistory() {return new NullPaymentHistory();}
```

class NullPaymentHistory...

```
get weeksDelinquentInLastYear() {return 0;}
```

客户端...

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinqu
```

我继续查看客户端代码，寻找是否有能用多态行为取代的地方。但也会有例外情况——客户端不想使用特例对象提供的逻辑，而是想做一些别的处理。我可能有23处客户端代码用"occupant"作为未知顾客的名字，但还有一处用了别的值。

客户端...

```
const name = ! isUnknown(aCustomer) ? aCustomer.name : "unkno
```

这种情况下，我只能在客户端保留特例检查的逻辑。我会对其做些修改，让它使用aCustomer对象身上的isUnknown函数，也就是对全局的isUnknown函数使用内联函数（115）。

客户端...

```
const name = aCustomer.isUnknown ? "unknown occupant" : aCust
```

处理完所有客户端代码后，全局的`isUnknown`函数应该没人再调用了，可以用移除死代码（237）将其移除。

范例：使用对象字面量

我们在上面处理的其实是一些很简单的值，却要创建一个这样的类，未免有点儿大动干戈。但在上面这个例子中，我必须创建这样一个类，因为`Customer`类是允许使用者更新其内容的。但如果面对一个只读的数据结构，我就可以改用字面量对象（literal object）。

还是前面这个例子——几乎完全一样，除了一件事：这次没有客户端对`Customer`对象做更新操作：

class Site...

```
get customer() {return this._customer;}
```

class Customer...

```
get name()            {...}
get billingPlan()     {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

客户端1...

```
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

客户端2...

```
const plan = (aCustomer === "unknown") ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

客户端3...

```
const weeksDelinquent = (aCustomer === "unknown") ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

和前面的例子一样，我首先在Customer中添加isUnknown属性，并创建一个包含同名字段的特例对象。这次的区别在于，特例对象是一个字面量。

class Customer...

```
get isUnknown() {return false;}
```

顶层作用域...

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
  };
}
```

然后我对检查特例的条件逻辑运用提炼函数（106）。

```
function isUnknown(arg) {
  return (arg === "unknown");
}
```

客户端1...

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

客户端2...

```
const plan = isUnknown(aCustomer) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

客户端3...

```
const weeksDelinquent = isUnknown(aCustomer) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

修改`Site`类和做条件判断的`isUnknown`函数，开始使用特例对象。

class Site...

```
get customer() {
  return (this._customer === "unknown") ? createUnknownCustom
}
```

顶层作用域...

```
function isUnknown(arg) {
  return arg.isUnknown;
}
```

然后把“以标准方式应对特例”的地方都替换成使用特例字面量的值。首先从“名字”开始：

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
  };
}
```

客户端1...

```
const customerName = aCustomer.name;
```

接着是“计价套餐”：

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };
}
```

客户端2...

```
const plan = aCustomer.billingPlan;
```

同样，我可以在字面量对象中创建一个嵌套的空支付记录对象：

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    },
  };
}
```

客户端3...

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinqu
```

如果使用了这样的字面量，应该使用诸如Object.freeze的方法将其冻结，使其不可变。通常，我还是喜欢用类多一点。

范例：使用变换

前面两个例子都涉及了一个类，其实本重构手法也同样适用于记录，只要增加一个变换步骤即可。

假设我们的输入是一个简单的记录结构，大概像这样：

```
{
  name: "Acme Boston",
  location: "Malden MA",
  // more site details
  customer: {
    name: "Acme Industries",
    billingPlan: "plan-451",
    paymentHistory: {
      weeksDelinquentInLastYear: 7
      //more
    },
    // more
  }
}
```

有时顾客的名字未知，此时标记的方式与前面一样：将customer字段标记为字符串"unknown"。

```
{
  name: "Warehouse Unit 15",
  location: "Malden MA",
  // more site details
  customer: "unknown",
}
```

客户端代码也类似，会检查"未知顾客"的情况：

客户端1...

```
const site = acquireSiteData();
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

客户端2...

```
const plan = (aCustomer === "unknown") ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

客户端3...

```
const weeksDelinquent = (aCustomer === "unknown") ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

我首先要让Site数据结构经过一次变换，目前变换中只做了深复制，没有对数据做任何处理。

客户端1...

```
  const rawSite = acquireSiteData();
  const site = enrichSite(rawSite);
  const aCustomer = site.customer;
  // ... lots of intervening code ...
  let customerName;
  if (aCustomer === "unknown") customerName = "occupant";
  else customerName = aCustomer.name;

function enrichSite(inputSite) {
  return _.cloneDeep(inputSite);
}
```

然后对“检查未知顾客”的代码运用提炼函数（106）。

```
function isUnknown(aCustomer) {
  return aCustomer === "unknown";
}
```

客户端1...

```
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

客户端2...

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

客户端3...

```
const weeksDelinquent = (isUnknown(aCustomer)) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

然后开始对Site数据做增强，首先是给customer字段加上isUnknown属性。

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
  };
```

```
if (isUnknown(result.customer)) result.customer = unknownCust
  else result.customer.isUnknown = false;
  return result;
}
```

随后修改检查特例的条件逻辑，开始使用新的属性。原来的检查逻辑也保留不动，所以现在的检查逻辑应该既能应对原来的Site数据，也能应对增强后的Site数据。

```
function isUnknown(aCustomer) {
  if (aCustomer === "unknown") return true;
  else return aCustomer.isUnknown;
}
```

测试，确保一切正常，然后针对特例使用函数组合成变换（149）。首先把“未知顾客的名字”的处理逻辑搬进增强函数。

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
  };

if (isUnknown(result.customer)) result.customer = unknownCust
  else result.customer.isUnknown = false;
  return result;
}
```

客户端1...

```
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// ... lots of intervening code ...
const customerName = aCustomer.name;
```

测试，然后是“未知顾客的计价套餐”的处理逻辑。

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };


if (isUnknown(result.customer)) result.customer = unknownCust
  else result.customer.isUnknown = false;
  return result;
}
```

客户端2...

```
const plan = aCustomer.billingPlan;
```

再次测试，然后处理最后一处客户端代码。

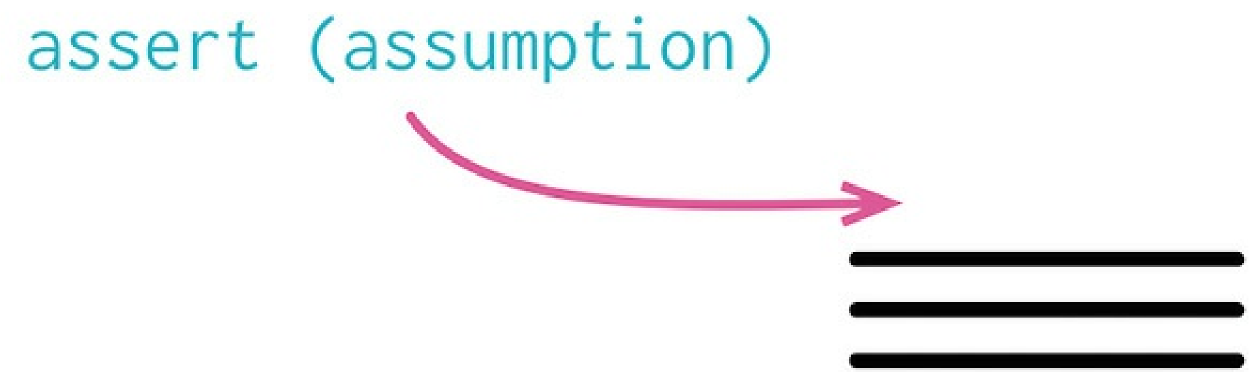
```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    }
  };


if (isUnknown(result.customer)) result.customer = unknownCust
  else result.customer.isUnknown = false;
  return result;
}
```

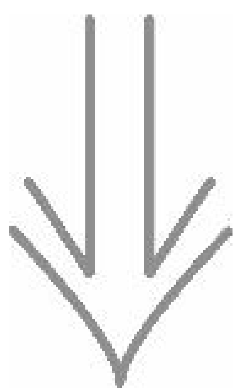
客户端3...

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinqu
```

10.6　引入断言（Introduce Assertion）




```
if (this.discountRate)
  base = base - (this.discountRate * base);
```



```
assert(this.discountRate>= 0);
if (this.discountRate)
  base = base - (this.discountRate * base);
```

动机

常常会有这样一段代码：只有当某个条件为真时，该段代码才能正常运行。例如，平方根计算只对正值才能进行，又例如，某个对象可能假设一组字段中至少有一个不等于null。

这样的假设通常并没有在代码中明确表现出来，你必须阅读整个算法才能看出。有时程序员会以注释写出这样的假设，而我要介绍的是一种更好的技术——使用断言明确标明这些假设。

断言是一个条件表达式，应该总是为真。如果它失败，表示程序员犯了错误。断言的失败不应该被系统任何地方捕捉。整个程序的行为在有没有断言出现的时候都应该完全一样。实际上，有些编程语言中的断言可以在编译期用一个开关完全禁用掉。

我常看见有人鼓励用断言来发现程序中的错误。这固然是一件好事，但却不是使用断言的唯一理由。断言是一种很有价值的交流形式——它们告诉阅读者，程序在执行到这一点时，对当前状态做了何种假设。另外断言对调试也很有帮助。而且，因为它们在交流上很有价值，即使解决了当下正在

追踪的错误，我还是倾向于把断言留着。自测试的代码降低了断言在调试方面的价值，因为逐步逼近的单元测试通常能更好地帮助调试，但我仍然看重断言在交流方面的价值。

做法

- 如果你发现代码假设某个条件始终为真，就加入一个断言明确说明这种情况。

因为断言应该不会对系统运行造成任何影响，所以“加入断言”永远都应该是行为保持的。

范例

下面是一个简单的例子：折扣。顾客（customer）会获得一个折扣率（discount rate），可以用于所有其购买的商品。

class Customer...

```
applyDiscount(aNumber) {
  return (this.discountRate)
    ? aNumber - (this.discountRate * aNumber)
    : aNumber;
}
```

这里有一个假设：折扣率永远是正数。我可以用断言明确标示出这个假设。但在一个三元表达式中没办法很简单地插入断言，所以我首先要把这个表达式转换成`if-else`的形式。

class Customer...

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else return aNumber - (this.discountRate * aNumber);
}
```

现在我就可以轻松地加入断言了。

class Customer...

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else {
    assert(this.discountRate >= 0);
    return aNumber - (this.discountRate * aNumber);
  }
}
```

对这个例子而言，我更愿意把断言放在设值函数上。如果在applyDiscount函数处发生断言失败，我还得先费力搞清楚非法的折扣率值起初是从哪儿放进去的。

class Customer...

```
set discountRate(aNumber) {
  assert(null === aNumber || aNumber >= 0);
  this._discountRate = aNumber;
}
```

真正引起错误的源头有可能很难发现——也许是输入数据中误写了一个减号，也许是某处代码做数据转换时犯了错误。像这样的断言对于发现错误源头特别有帮助。

注意，不要滥用断言。我不会使用断言来检查所有“我认为应该为真”的条件，只用来检查“必须为真”的条件。滥用断言可能会造成代码重复，尤其是在处理上面这样的条件逻辑时。所以我发现，很有必要去掉条件逻辑中的重复，通常可以借助提炼函数（106）手法。

我只用断言预防程序员的错误。如果要从某个外部数据源读取数据，那么所有对输入值的检查都

应该是程序的一等公民，而不能用断言实现——除非我对这个外部数据源有绝对的信心。断言是帮助我们跟踪bug的最后一招，所以，或许听来讽刺，只有当我认为断言绝对不会失败的时候，我才会使用断言。

第11章　重构API

模块和函数是软件的骨肉，而API则是将骨肉连接起来的关节。易于理解和使用的API非常重要，但同时也很难获得。随着对软件理解的加深，我会学到如何改进API，这时我便需要对API进行重构。

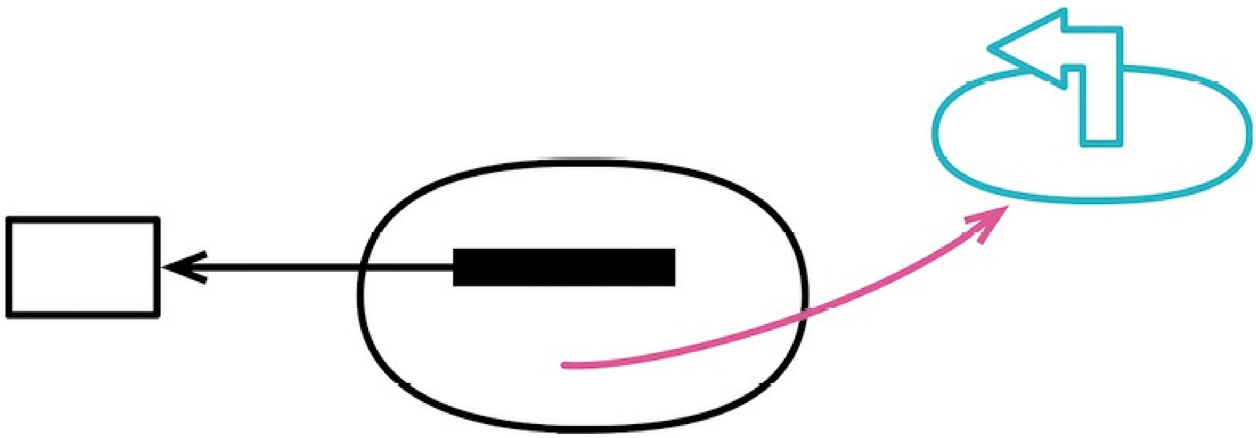
好的API会把更新数据的函数与只是读取数据的函数清晰分开。如果我看到这两类操作被混在一起，就会用将查询函数和修改函数分离（306）将它们分开。如果两个函数的功能非常相似、只有一些数值不同，我可以用函数参数化（310）将其统一。但有些参数其实只是一个标记，根据这个标记的不同，函数会有截然不同的行为，此时最好用移除标记参数（314）将不同的行为彻底分开。

在函数间传递时，数据结构常会毫无必要地被拆开，我更愿意用保持对象完整（319）将其聚拢。函数需要的一份信息，究竟何时应该作为参数传入、何时应该调用一个函数获得，这是一个需要反复推敲的决定，推敲的过程中常常要用到以查询取代参数（324）和以参数取代查询（327）。

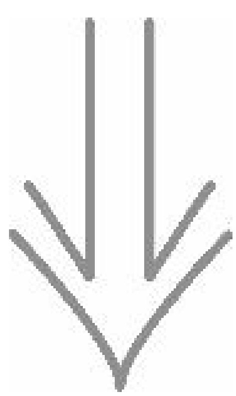
类是一种常见的模块形式。我希望尽可能保持对象不可变，所以只要有可能，我就会使用移除设值函数（331）。当调用者要求一个新对象时，我经常需要比构造函数更多的灵活性，可以借助以工厂函数取代构造函数（334）获得这种灵活性。

有时你会遇到一个特别复杂的函数，围绕着它传入传出一大堆数据。最后两个重构手法专门用于破解这个难题。我可以用以命令取代函数（337）将这个函数变成对象，这样对函数体使用提炼函数（106）时会更容易。如果稍后我对该函数做了简化，不再需要将其作为命令对象了，可以用以函数取代命令（344）再把它变回函数。

11.1　将查询函数和修改函数分离（Separate Query from Modifier）



```
function getTotalOutstandingAndSendBill() {
  const result = customer.invoices.reduce((total, each) => ea
  sendBill();
  return result;
}
```



```
function totalOutstanding() {
  return customer.invoices.reduce((total, each) => each.amoun
```

```
}
function sendBill() {
  emailGateway.send(formatBill(customer));
}
```

动机

如果某个函数只是提供一个值，没有任何看得到的副作用，那么这是一个很有价值的东西。我可以任意调用这个函数，也可以把调用动作搬到调用函数的其他地方。这种函数的测试也更容易。简而言之，需要操心的事情少多了。

明确表现出“有副作用”与“无副作用”两种函数之间的差异，是个很好的想法。下面是一条好规则：任何有返回值的函数，都不应该有看得到的副作用——命令与查询分离（Command-Query Separation）[mf-cqs]。有些程序员甚至将此作为一条必须遵守的规则。就像对待任何东西一样，我并不绝对遵守它，不过我总是尽量遵守，而它也回报我很好的效果。

如果遇到一个“既有返回值又有副作用”的函数，我就会试着将查询动作从修改动作中分离出来。

你也许已经注意到了：我使用“看得到的副作用”这种说法。有一种常见的优化办法是：将查询所得结果缓存于某个字段中，这样一来后续的重复查询就可以大大加快速度。虽然这种做法改变了对象中缓存的状态，但这一修改是察觉不到的，因为不论如何查询，总是获得相同结果。

做法

- 复制整个函数，将其作为一个查询来命名。

  如果想不出好名字，可以看看函数返回的是什么。查询的结果会被填入一个变量，这个变量的名字应该能对函数如何命名有所启发。

- 从新建的查询函数中去掉所有造成副作用的语句。
- 执行静态检查。
- 查找所有调用原函数的地方。如果调用处用到了该函数的返回值，就将其改为调用新建的查询函数，并在下面马上再调用一次原函数。每

次修改之后都要测试。

- 从原函数中去掉返回值。
- 测试。

完成重构之后，查询函数与原函数之间常会有重复代码，可以做必要的清理。

范例

有这样一个函数：它会遍历一份恶棍（miscreant）名单，检查一群人（people）里是否混进了恶棍。如果发现了恶棍，该函数会返回恶棍的名字，并拉响警报。如果人群中有多名恶棍，该函数也只汇报找出的第一名恶棍（我猜这就已经够了）。

```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

首先我复制整个函数，用它的查询部分功能为其命名。

```
function findMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

然后在新建的查询函数中去掉副作用。

```
function findMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

然后找到所有原函数的调用者，将其改为调用新建的查询函数，并在其后调用一次修改函数（也

就是原函数）。于是代码

```
const found = alertForMiscreant(people);
```

就变成了

```
const found = findMiscreant(people);
alertForMiscreant(people);
```

现在可以从修改函数中去掉所有返回值了。

```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return;
    }
    if (p === "John") {
      setOffAlarms();
      return;
    }
  }
  return;
}
```

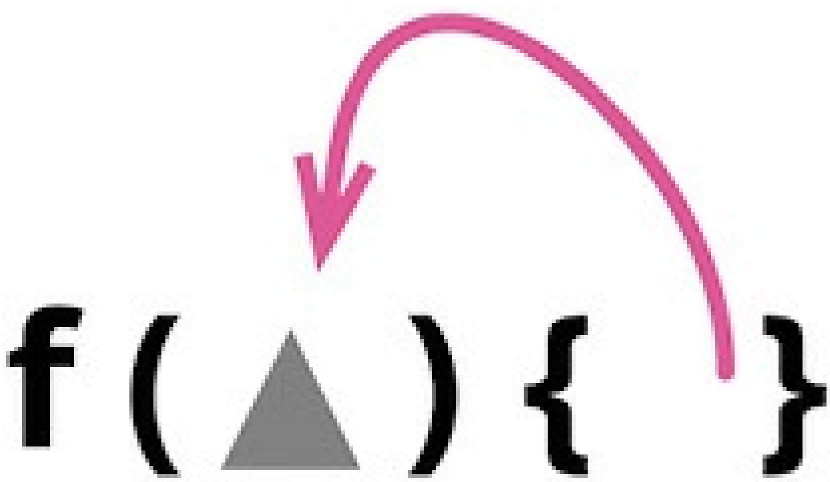
现在，原来的修改函数和新建的查询函数之间有大量的重复代码，我可以使用替换算法（195），让修改函数使用查询函数。

```
function alertForMiscreant (people) {
  if (findMiscreant(people) !== "") setOffAlarms();
```

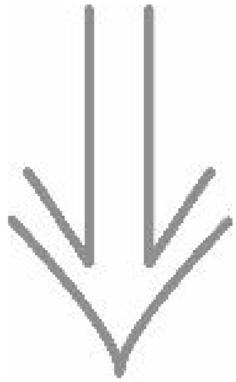
```
}
```

11.2　函数参数化（Parameterize Function）

曾用名：令函数携带参数（Parameterize Method）




```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```



```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

动机

如果我发现两个函数逻辑非常相似，只有一些字面量值不同，可以将其合并成一个函数，以参数的形式传入不同的值，从而消除重复。这个重构可以使函数更有用，因为重构后的函数还可以用于处理其他的值。

做法

- 从一组相似的函数中选择一个。

- 运用改变函数声明（124），把需要作为参数传入的字面量添加到参数列表中。
- 修改该函数所有的调用处，使其在调用时传入该字面量值。
- 测试。
- 修改函数体，令其使用新传入的参数。每使用一个新参数都要测试。
- 对于其他与之相似的函数，逐一将其调用处改为调用已经参数化的函数。每次修改后都要测试。

如果第一个函数经过参数化以后不能直接替代另一个与之相似的函数，就先对参数化之后的函数做必要的调整，再做替换。

范例

下面是一个显而易见的例子：

```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
```

```
}
```

很明显我可以用下面这个函数来替换上面两个：

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

情况可能比这个更复杂一些。例如下列代码：

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
       bottomBand(usage) * 0.03
       + middleBand(usage) * 0.05
       + topBand(usage) * 0.07;
  return usd(amount);
}

function bottomBand(usage) {
  return Math.min(usage, 100);
}

function middleBand(usage) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}

function topBand(usage) {
  return usage > 200 ? usage - 200 : 0;
}
```

这几个函数中的逻辑明显很相似，但是不是相似到足以支撑一个参数化的计算“计费档

次”（band）的函数？这次就不像前面第一个例子那样一目了然了。

在尝试对几个相关的函数做参数化操作时，我会先从中挑选一个，在上面添加参数，同时留意其他几种情况。在类似这样处理“范围”的情况下，通常从位于中间的范围开始着手较好。所以我首先选择了middleBand函数来添加参数，然后调整其他的调用者来适应它。

middleBand使用了两个字面量值，即100和200，分别代表“中间档次”的下界和上界。我首先用改变函数声明（124）加上这两个参数，同时顺手给函数改个名，使其更好地表述参数化之后的含义。

```
function withinBand(usage, bottom, top) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}

function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
       bottomBand(usage) * 0.03
       + withinBand(usage, 100, 200) * 0.05
       + topBand(usage) * 0.07;
  return usd(amount);
}
```

在函数体内部，把一个字面量改为使用新传入的参数：

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, 200) - bottom : 0;
}
```

然后是另一个：

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, top) - bottom : 0;
}
```

对于原本调用bottomBand函数的地方，我将其改为调用参数化了的新函数。

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
       withinBand(usage, 0, 100) * 0.03
       + withinBand(usage, 100, 200) * 0.05
       + topBand(usage) * 0.07;
  return usd(amount);
}

function bottomBand(usage) {
  return Math.min(usage, 100);
}
```

为了替换对topBand的调用，我就得用代表“无穷大”的Infinity作为这个范围的上界。

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
       withinBand(usage, 0, 100) * 0.03
```

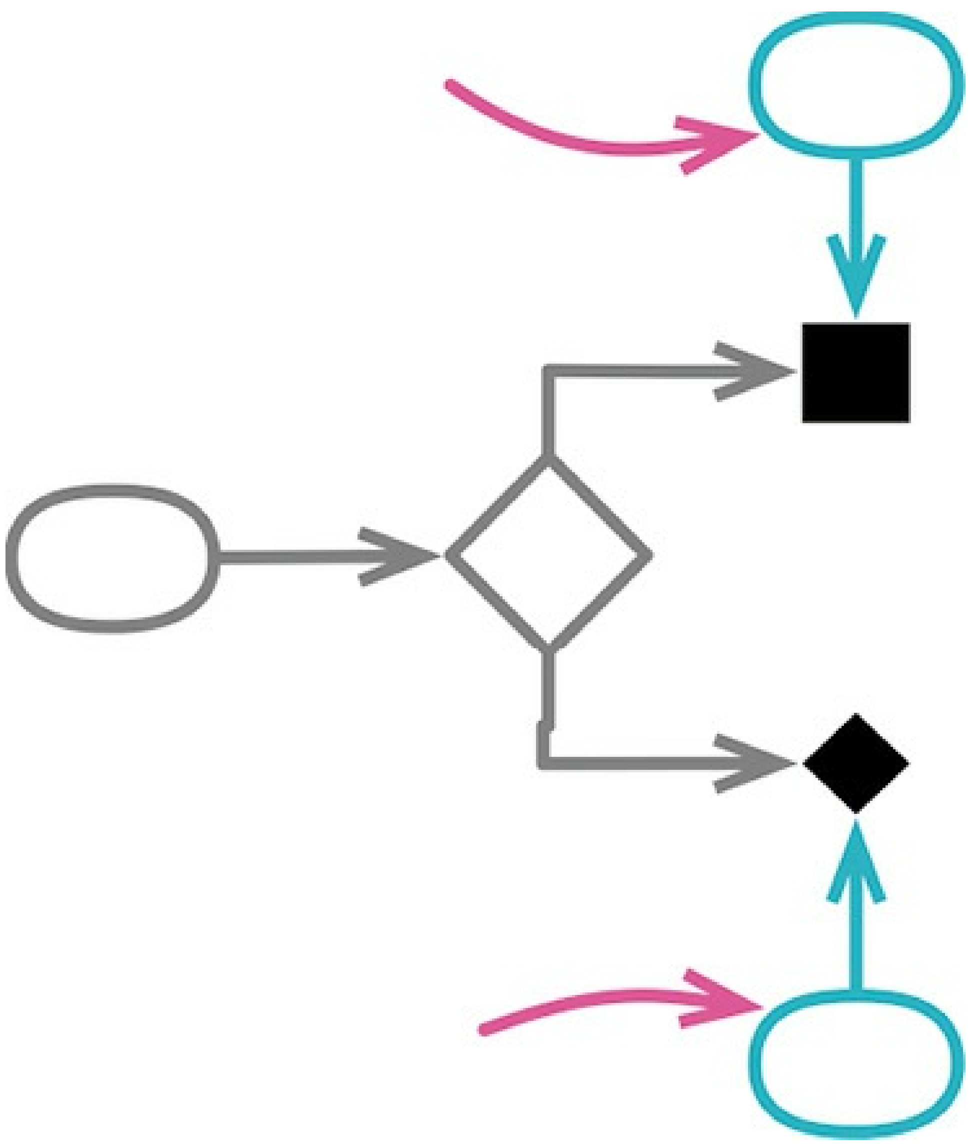
```
        + withinBand(usage, 100, 200) * 0.05
        + withinBand(usage, 200, Infinity) * 0.07;
  return usd(amount);
}

function topBand(usage) {
    return usage > 200 ? usage - 200 : 0;
}
```

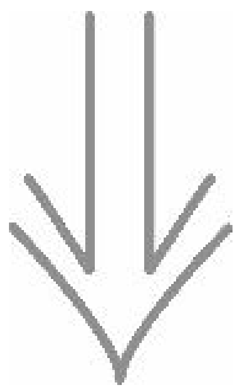
照现在的逻辑，baseCharge一开始的卫语句已经可以去掉了。不过，尽管这条语句已经失去了逻辑上的必要性，我还是愿意把它留在原地，因为它阐明了“传入的usage参数为负数”这种情况是如何处理的。

11.3　移除标记参数（Remove Flag Argument）

曾用名：以明确函数取代参数（Replace Parameter with Explicit Methods）



```
function setDimension(name, value) {
  if (name === "height") {
    this._height = value;
    return;
  }
  if (name === "width") {
    this._width = value;
    return;
  }
}
```



```
function setHeight(value) {this._height = value;}
function setWidth (value) {this._width = value;}
```

动机

“标记参数”是这样的一种参数：调用者用它来指示被调函数应该执行哪一部分逻辑。例如，我可能有下面这样一个函数：

```
function bookConcert(aCustomer, isPremium) {
```

```
  if (isPremium) {
    // logic for premium booking
  } else {
    // logic for regular booking
  }
}
```

要预订一场高级音乐会（premium concert），就得这样发起调用：

```
bookConcert(aCustomer, true);
```

标记参数也可能以枚举的形式出现：

```
bookConcert(aCustomer, CustomerType.PREMIUM);
```

或者是以字符串（或者符号，如果编程语言支持的话）的形式出现：

```
bookConcert(aCustomer, "premium");
```

我不喜欢标记参数，因为它们让人难以理解到底有哪些函数可以调用、应该怎么调用。拿到一份API以后，我首先看到的是一系列可供调用的函数，但标记参数却隐藏了函数调用中存在的差异性。使用这样的函数，我还得弄清标记参数有哪些

可用的值。布尔型的标记尤其糟糕，因为它们不能清晰地传达其含义——在调用一个函数时，我很难弄清true到底是什么意思。如果明确用一个函数来完成一项单独的任务，其含义会清晰得多。

```
premiumBookConcert(aCustomer);
```

并非所有类似这样的参数都是标记参数。如果调用者传入的是程序中流动的数据，这样的参数不算标记参数；只有调用者直接传入字面量值，这才是标记参数。另外，在函数实现内部，如果参数值只是作为数据传给其他函数，这就不是标记参数；只有参数值影响了函数内部的控制流，这才是标记参数。

移除标记参数不仅使代码更整洁，并且能帮助开发工具更好地发挥作用。去掉标记参数后，代码分析工具能更容易地体现出“高级”和“普通”两种预订逻辑在使用时的区别。

如果一个函数有多个标记参数，可能就不得不将其保留，否则我就得针对各个参数的各种取值的所有组合情况提供明确函数。不过这也是一个信号，说明这个函数可能做得太多，应该考虑是否能用更简单的函数来组合出完整的逻辑。

做法

- 针对参数的每一种可能值，新建一个明确函数。

如果主函数有清晰的条件分发逻辑，可以用分解条件表达式（260）创建明确函数；否则，可以在原函数之上创建包装函数。

- 对于“用字面量值作为参数”的函数调用者，将其改为调用新建的明确函数。

范例

在浏览代码时，我发现多处代码在调用一个函数计算物流（shipment）的到货日期（delivery date）。一些调用代码类似这样：

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

另一些调用代码则是这样：

```
aShipment.deliveryDate = deliveryDate(anOrder, false);
```

面对这样的代码，我立即开始好奇：参数里这个布尔值是什么意思？是用来干什么的？

deliveryDate函数主体如下所示：

```
function deliveryDate(anOrder, isRush) {
  if (isRush) {
    let deliveryTime;

if (["MA", "CT"]     .includes(anOrder.deliveryState)) deliver

else if (["NY", "NH"].includes(anOrder.deliveryState)) delive
    else deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
  }
  else {
    let deliveryTime;

if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliv

else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliv
    else deliveryTime = 4;
    return anOrder.placedOn.plusDays(2 + deliveryTime);
  }
}
```

原来调用者用这个布尔型字面量来判断应该运行哪个分支的代码——典型的标记参数。然而函数的重点就在于要遵循调用者的指令，所以最好是用明确函数的形式明确说出调用者的意图。

对于这个例子，我可以使用分解条件表达式（260），得到下列代码：

```
function deliveryDate(anOrder, isRush) {
  if (isRush) return rushDeliveryDate(anOrder);
  else       return regularDeliveryDate(anOrder);
}
function rushDeliveryDate(anOrder) {
  let deliveryTime;

if (["MA", "CT"]     .includes(anOrder.deliveryState)) delive

else if (["NY", "NH"].includes(anOrder.deliveryState)) delive
  else deliveryTime = 3;
  return anOrder.placedOn.plusDays(1 + deliveryTime);
}
function regularDeliveryDate(anOrder) {
  let deliveryTime;

if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliv

else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliv
  else deliveryTime = 4;
  return anOrder.placedOn.plusDays(2 + deliveryTime);
}
```

这两个函数能更好地表达调用者的意图，现在我可以修改调用方代码了。调用代码

```
aShipment.deliveryDate = deliveryDate(anOrder,true);
```

可以改为

```
aShipment.deliveryDate = rushDeliveryDate(anOrder);
```

另一个分支也类似。

处理完所有调用处，我就可以移除deliveryDate函数。

这个参数是标记参数，不仅因为它是布尔类型，而且还因为调用方以字面量的形式直接设置参数值。如果所有调用deliveryDate的代码都像这样：

```
const isRush = determineIfRush(anOrder);
aShipment.deliveryDate = deliveryDate(anOrder, isRush);
```

那我对这个函数的签名没有任何意见（不过我还是想用分解条件表达式（260）清理其内部实现）。

可能有一些调用者给这个参数传入的是字面量，将其作为标记参数使用；另一些调用者则传入正常的数据。若果真如此，我还是会使用移除标记参数（314），但不修改传入正常数据的调用者，重构结束时也不删除deliveryDate函数。这样我就提供了两套接口，分别支持不同的用途。

直接拆分条件逻辑是实施本重构的好方法，但只有当“根据参数值做分发”的逻辑发生在函数最外层（或者可以比较容易地将其重构至函数最外层）

的时候，这一招才好用。函数内部也有可能以一种更纠结的方式使用标记参数，例如下面这个版本的deliveryDate函数：

```
function deliveryDate(anOrder, isRush) {
  let result;
  let deliveryTime;

if (anOrder.deliveryState === "MA" || anOrder.deliveryState =
    deliveryTime = isRush? 1 : 2;

else if (anOrder.deliveryState === "NY" || anOrder.deliverySt
    deliveryTime = 2;
    if (anOrder.deliveryState === "NH" && !isRush)
      deliveryTime = 3;
  }
  else if (isRush)
    deliveryTime = 3;
  else if (anOrder.deliveryState === "ME")
    deliveryTime = 3;
  else
    deliveryTime = 4;
  result = anOrder.placedOn.plusDays(2 + deliveryTime);
  if (isRush) result = result.minusDays(1);
  return result;
}
```

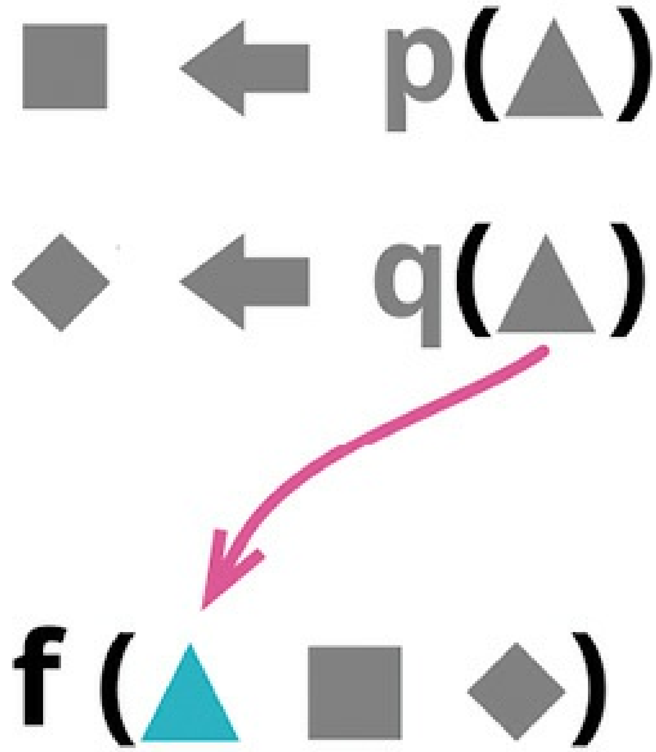
这种情况下，想把围绕isRush的分发逻辑剥离到顶层，需要的工作量可能会很大。所以我选择退而求其次，在deliveryDate之上添加两个函数：

```
function rushDeliveryDate (anOrder) {return deliveryDate(anOr
function regularDeliveryDate(anOrder) {return deliveryDate(an
```

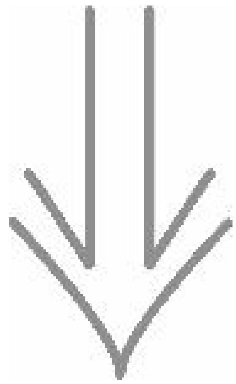
本质上，这两个包装函数分别代表了deliveryDate函数一部分的使用方式。不过它们并非从原函数中拆分而来，而是用代码文本强行定义的。

随后，我同样可以逐一替换原函数的调用者，就跟前面分解条件表达式之后的处理一样。如果没有任何一个调用者向isRush参数传入正常的数据，我最后会限制原函数的可见性，或是将其改名（例如改为deliveryDateHelperOnly），让人一见即知不应直接使用这个函数。

11.4　保持对象完整（Preserve Whole Object）



```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (aPlan.withinRange(low, high))
```



```
if (aPlan.withinRange(aRoom.daysTempRange))
```

动机

如果我看见代码从一个记录结构中导出几个值，然后又把这几个值一起传递给一个函数，我会更愿意把整个记录传给这个函数，在函数体内部导出所需的值。

“传递整个记录”的方式能更好地应对变化：如果将来被调的函数需要从记录中导出更多的数据，我就不用为此修改参数列表。并且传递整个记录也能缩短参数列表，让函数调用更容易看懂。如果有很多函数都在使用记录中的同一组数据，处理这部分数据的逻辑常会重复，此时可以把这些处理逻辑搬移到完整对象中去。

也有时我不想采用本重构手法，因为我不想让被调函数依赖完整对象，尤其是在两者不在同一个模块中的时候。

从一个对象中抽取出几个值，单独对这几个值做某些逻辑操作，这是一种代码坏味道（依恋情结），通常标志着这段逻辑应该被搬移到对象中。保持对象完整经常发生在引入参数对象（140）之后，我会搜寻使用原来的数据泥团的代码，代之以使用新的对象。

如果几处代码都在使用对象的一部分功能，可能意味着应该用提炼类（182）把这一部分功能单独提炼出来。

还有一种常被忽视的情况：调用者将自己的若干数据作为参数，传递给被调用函数。这种情况下，我可以将调用者的自我引用（在JavaScript中就是`this`）作为参数，直接传递给目标函数。

做法

- 新建一个空函数，给它以期望中的参数列表（即传入完整对象作为参数）。

给这个函数起一个容易搜索的名字，这样到重构结束时方便替换。

- 在新函数体内调用旧函数，并把新的参数（即完整对象）映射到旧的参数列表（即来源于完整对象的各项数据）。
- 执行静态检查。
- 逐一修改旧函数的调用者，令其使用新函数，每次修改之后执行测试。

修改之后，调用处用于“从完整对象中导出参数值”的代码可能就没用了，可以用移除死代码（237）去掉。

- 所有调用处都修改过来之后，使用内联函数（115）把旧函数内联到新函数体内。
- 给新函数改名，从重构开始时的容易搜索的临时名字，改为使用旧函数的名字，同时修改所有调用处。

范例

我们想象一个室温监控系统，它负责记录房间一天中的最高温度和最低温度，然后将实际的温度范围与预先规定的温度控制计划（heating plan）相比较，如果当天温度不符合计划要求，就发出警告。

调用方...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("room temperature went outside range");
```

class HeatingPlan...

```
withinRange(bottom, top) {

return (bottom >= this._temperatureRange.low) && (top <= this
}
```

其实我不必将“温度范围”的信息拆开来单独传递，只需将整个范围对象传递给`withinRange`函数即可。

首先，我在`HeatingPlan`类中新添一个空函数，给它赋予我认为合理的参数列表。

class HeatingPlan...

```
xxNEWwithinRange(aNumberRange) {
}
```

因为这个函数最终要取代现有的withinRange函数，所以它也用了同样的名字，再加上一个容易替换的前缀。

然后在新函数体内调用现有的withinRange函数。因此，新函数体就完成了从新参数列表到旧函数参数列表的映射。

class HeatingPlan...

```
xxNEWwithinRange(aNumberRange) {
  return this.withinRange(aNumberRange.low, aNumberRange.high
}
```

现在开始正式的替换工作了，我要找到调用现有函数的地方，将其改为调用新函数。

调用方...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

在修改调用处时，我可能会发现一些代码在修改后已经不再需要，此时可以使用移除死代码（237）。

调用方...

```
~~const low = aRoom.daysTempRange.low;~~
~~const high = aRoom.daysTempRange.high;~~
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

每次替换一处调用代码，每次修改后都要测试。

调用处全部替换完成后，用内联函数（115）将旧函数内联到新函数体内。

class HeatingPlan...

```
xxNEWwithinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
    (aNumberRange.high <= this._temperatureRange.high);
}
```

终于可以去掉新函数那难看的前缀了，记得同时修改所有调用者。就算我所使用的开发环境不支持可靠的函数改名操作，有这个极具特色的前缀在，我也可以很方便地全局替换。

class HeatingPlan...

```
withinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
    (aNumberRange.high <= this._temperatureRange.high);
}
```

调用方...

```
if (!aPlan.withinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

范例：换个方式创建新函数

在上面的示例中，我直接编写了新函数。大多数时候，这一步非常简单，也是创建新函数最容易的方式。不过有时还会用到另一种方式：可以完全

通过重构手法的组合来得到新函数。

我从一处调用现有函数的代码开始。

调用方...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("room temperature went outside range");
```

我要先对代码做一些整理，以便用提炼函数（106）来创建新函数。目前的调用者代码还不具备可提炼的函数雏形，不过我可以先做几次提炼变量（119），使其轮廓显现出来。首先，我要把对旧函数的调用从条件判断中解放出来。

调用方...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

然后把输入参数也提炼出来。

调用方...

```
const tempRange = aRoom.daysTempRange;
const low = tempRange.low;
const high = tempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

完成这一步之后，就可以用提炼函数（106）来创建新函数。

调用方...

```
const tempRange = aRoom.daysTempRange;
const isWithinRange = xxNEWwithinRange(aPlan, tempRange);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

顶层作用域...

```
function xxNEWwithinRange(aPlan, tempRange) {
  const low = tempRange.low;
  const high = tempRange.high;
  const isWithinRange = aPlan.withinRange(low, high);
  return isWithinRange;
}
```

由于旧函数属于另一个上下文（HeatingPlan类），我需要用搬移函数（198）把新函数也搬过去。

调用方...

```
const tempRange = aRoom.daysTempRange;
const isWithinRange = aPlan.xxNEWwithinRange(tempRange);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

class HeatingPlan...

```
xxNEWwithinRange(tempRange) {
  const low = tempRange.low;
  const high = tempRange.high;
  const isWithinRange = this.withinRange(low, high);
  return isWithinRange;
}
```

剩下的过程就跟前面一样了：替换其他调用者，然后把旧函数内联到新函数中。重构刚开始的时候，为了清晰分离函数调用，以便提炼出新函数，我提炼了几个变量出来，现在可以把这些变量也内联回去。

这种方式的好处在于：它完全是由其他重构手

法组合而成的。如果我使用的开发工具支持可靠的提炼和内联操作，用这种方式进行本重构会特别流畅。

11.5　以查询取代参数（Replace Parameter with Query）

曾用名：以函数取代参数（Replace　Parameter with Method）

反向重构：以参数取代查询（327）




```
availableVacation(anEmployee, anEmployee.grade);

function availableVacation(anEmployee, grade) {
  // calculate vacation...
```

```
availableVacation(anEmployee)

function availableVacation(anEmployee) {
  const grade = anEmployee.grade;
  // calculate vacation...
```

动机

函数的参数列表应该总结该函数的可变性，标示出函数可能体现出行为差异的主要方式。和任何代码中的语句一样，参数列表应该尽量避免重复，并且参数列表越短就越容易理解。

如果调用函数时传入了一个值，而这个值由函数自己来获得也是同样容易，这就是重复。这个本不必要的参数会增加调用者的难度，因为它不得不找出正确的参数值，其实原本调用者是不需要费这个力气的。

“同样容易”四个字，划出了一条判断的界限。去除参数也就意味着“获得正确的参数值”的责任被转移：有参数传入时，调用者需要负责获得正确的参数值；参数去除后，责任就被转移给了函数本身。一般而言，我习惯于简化调用方，因此我愿意把责任移交给函数本身，但如果函数难以承担这份责任，就另当别论了。

不使用以查询取代参数最常见的原因是，移除参数可能会给函数体增加不必要的依赖关系——迫使函数访问某个程序元素，而我原本不想让函数了解这个元素的存在。这种“不必要的依赖关系”除了新增的以外，也可能是我想要稍后去除的，例如为了去除一个参数，我可能会在函数体内调用一个有问题的函数，或是从一个对象中获取某些原本想要剥离出去的数据。在这些情况下，都应该慎重考虑使用以查询取代参数。

如果想要去除的参数值只需要向另一个参数查询就能得到，这是使用以查询取代参数最安全的场景。如果可以从一个参数推导出另一个参数，那么几乎没有任何理由要同时传递这两个参数。

另外有一件事需要留意：如果在处理的函数具有引用透明性（referential transparency，即，不论任何时候，只要传入相同的参数值，该函数的行为永远一致），这样的函数既容易理解又容易测试，

我不想使其失去这种优秀品质。我不会去掉它的参数，让它去访问一个可变的全局变量。

做法

- 如果有必要，使用提炼函数（106）将参数的计算过程提炼到一个独立的函数中。
- 将函数体内引用该参数的地方改为调用新建的函数。每次修改后执行测试。
- 全部替换完成后，使用改变函数声明（124）将该参数去掉。

范例

某些重构会使参数不再被需要，这是我最常用到以查询取代参数的场合。考虑下列代码。

class Order...

```
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  let discountLevel;
```

```
  if (this.quantity > 100) discountLevel = 2;
  else discountLevel = 1;
  return this.discountedPrice(basePrice, discountLevel);
}

discountedPrice(basePrice, discountLevel) {
  switch (discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

在简化函数逻辑时，我总是热衷于使用以查询取代临时变量（178），于是就得到了如下代码。

class Order...

```
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  return this.discountedPrice(basePrice, this.discountLevel);
}

get discountLevel() {
  return (this.quantity > 100) ? 2 : 1;
}
```

到这一步，已经不需要再把`discountLevel`的计算结果传给`discountedPrice`了，后者可以自己调用`discountLevel`函数，不会增加任何难度。

因此，我把`discountedPrice`函数中用到这个参数的地方全都改为直接调用`discountLevel`函

数。

class Order...

```
discountedPrice(basePrice, discountLevel) {
  switch (this.discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

然后用改变函数声明（124）手法移除该参数。

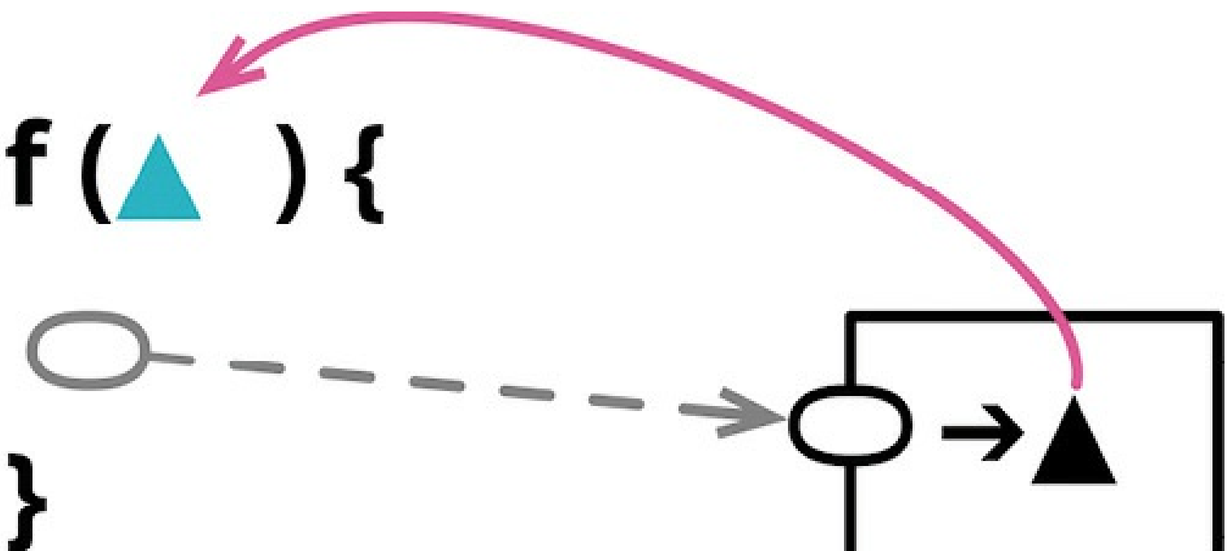
class Order...

```
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  return this.discountedPrice(basePrice, this.discountLevel);
}

discountedPrice(basePrice, discountLevel) {
  switch (this.discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

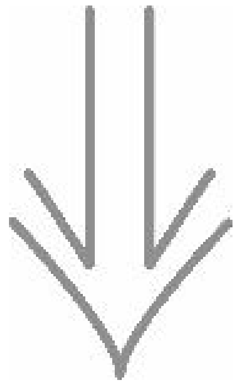
11.6 以参数取代查询（Replace Query with Parameter）

反向重构：以查询取代参数（324）




```
targetTemperature(aPlan)

function targetTemperature(aPlan) {
  currentTemperature = thermostat.currentTemperature;
  // rest of function...
```



```
targetTemperature(aPlan, thermostat.currentTemperature)

function targetTemperature(aPlan, currentTemperature) {
  // rest of function...
```

动机

在浏览函数实现时，我有时会发现一些令人不快的引用关系，例如，引用一个全局变量，或者引用另一个我想要移除的元素。为了解决这些令人不快的引用，我需要将其替换为函数参数，从而将处理引用关系的责任转交给函数的调用者。

需要使用本重构的情况大多源于我想要改变代码的依赖关系——为了让目标函数不再依赖于某个元素，我把这个元素的值以参数形式传递给该函数。这里需要注意权衡：如果把所有依赖关系都变成参数，会导致参数列表冗长重复；如果作用域之

间的共享太多，又会导致函数间依赖过度。我一向不善于微妙的权衡，所以“能够可靠地改变决定”就显得尤为重要，这样随着我的理解加深，程序也能从中受益。

如果一个函数用同样的参数调用总是给出同样的结果，我们就说这个函数具有“引用透明性”（referential transparency），这样的函数理解起来更容易。如果一个函数使用了另一个元素，而后者不具引用透明性，那么包含该元素的函数也就失去了引用透明性。只要把“不具引用透明性的元素”变成参数传入，函数就能重获引用透明性。虽然这样就把责任转移给了函数的调用者，但是具有引用透明性的模块能带来很多益处。有一个常见的模式：在负责逻辑处理的模块中只有纯函数，其外再包裹处理I/O和其他可变元素的逻辑代码。借助以参数取代查询，我可以提纯程序的某些组成部分，使其更容易测试、更容易理解。

不过以参数取代查询并非只有好处。把查询变成参数以后，就迫使调用者必须弄清如何提供正确的参数值，这会增加函数调用者的复杂度，而我在设计接口时通常更愿意让接口的消费者更容易使用。归根到底，这是关于程序中责任分配的问题，而这方面的决策既不容易，也不会一劳永逸——这就是我需要非常熟悉本重构（及其反向重构）的原

因。

做法

- 对执行查询操作的代码使用提炼变量（119），将其从函数体中分离出来。
- 现在函数体代码已经不再执行查询操作（而是使用前一步提炼出的变量），对这部分代码使用提炼函数（106）。

　　给提炼出的新函数起一个容易搜索的名字，以便稍后改名。

- 使用内联变量（123），消除刚才提炼出来的变量。
- 对原来的函数使用内联函数（115）。
- 对新函数改名，改回原来函数的名字。

范例

我们想象一个简单却又烦人的温度控制系统。用户可以从一个温控终端（thermostat）指定温度，但指定的目标温度必须在温度控制计划（heating plan）允许的范围内。

class HeatingPlan...

```
get targetTemperature() {
  if (thermostat.selectedTemperature > this._max) return this
  else if (thermostat.selectedTemperature < this._min) return
  else return thermostat.selectedTemperature;
}
```

调用方...

```
if    (thePlan.targetTemperature > thermostat.currentTemperat
else if(thePlan.targetTemperature<thermostat.currentTemperatu
else setOff();
```

系统的温控计划规则抑制了我的要求，作为这样一个系统的用户，我可能会感到很烦恼。不过作为程序员，我更担心的是`targetTemperature`函数依赖于全局的`thermostat`对象。我可以把需要这个对象提供的信息作为参数传入，从而打破对该对象的依赖。

首先，我要用提炼变量（119）把“希望作为参数传入的信息”提炼出来。

class HeatingPlan...

```
get targetTemperature() {

const selectedTemperature = thermostat.selectedTemperature;

if      (selectedTemperature > this._max) return this._max;

else if (selectedTemperature < this._min) return this._min;
  else return selectedTemperature;
}
```

这样可以比较容易地用提炼函数（106）把整个函数体提炼出来，只剩“计算参数值”的逻辑还在原地。

class HeatingPlan...

```
get targetTemperature() {

const selectedTemperature = thermostat.selectedTemperature;
  return this.xxNEWtargetTemperature(selectedTemperature);
}

xxNEWtargetTemperature(selectedTemperature) {

if      (selectedTemperature > this._max) return this._max;

else if (selectedTemperature < this._min) return this._min;
```

```
  else return selectedTemperature;
}
```

然后把刚才提炼出来的变量内联回去，于是旧函数就只剩一个简单的调用。

class HeatingPlan...

```
get targetTemperature() {
  return this.xxNEWtargetTemperature(thermostat.selectedTempe
}
```

现在可以对其使用内联函数（115）。

调用方...

```
if   (thePlan.xxNEWtargetTemperature(thermostat.selectedTempe
      thermostat.currentTemperature)
  setToHeat();
else if (thePlan.xxNEWtargetTemperature(thermostat.selectedTe
         thermostat.currentTemperature)
  setToCool();
else
  setOff();
```

再把新函数改名，用回旧函数的名字。得益于之前给它起了一个容易搜索的名字，现在只要把前

缀去掉就行。

调用方...

```
if   (thePlan.targetTemperature(thermostat.selectedTemperatur
     thermostat.currentTemperature)
  setToHeat();
else if (thePlan.targetTemperature(thermostat.selectedTempera
        thermostat.currentTemperature)
  setToCool();
else
  setOff();
```

class HeatingPlan...

```
targetTemperature(selectedTemperature) {

if      (selectedTemperature > this._max) return this._max;

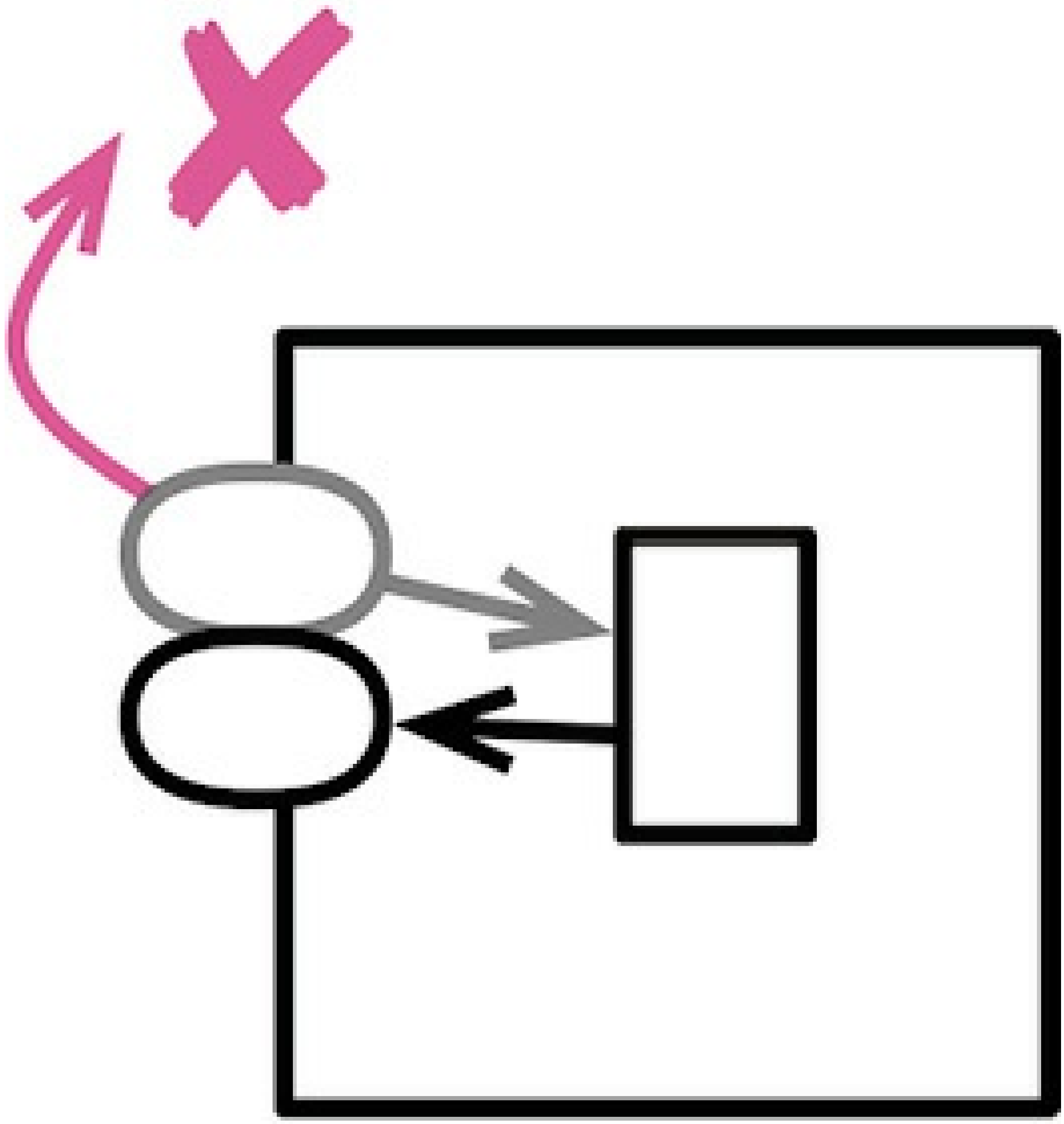
else if (selectedTemperature < this._min) return this._min;
  else return selectedTemperature;
}
```

调用方的代码看起来比重构之前更笨重了，这是使用本重构手法的常见情况。将一个依赖关系从一个模块中移出，就意味着将处理这个依赖关系的责任推回给调用者。这是为了降低耦合度而付出的代价。

但是，去除对thermostat对象的耦合，并不是本重构带来的唯一收益。HeatingPlan类本身是不可变的——字段的值都在构造函数中设置，任何函数都不会修改它们。（不用费心去查看整个类的代码，相信我就好。）在不可变的HeatingPlan基础上，把对thermostat的依赖移出函数体之后，我又使targetTemperature函数具备了引用透明性。从此以后，只要在同一个HeatingPlan对象上用同样的参数调用targetTemperature函数，我会始终得到同样的结果。如果HeatingPlan的所有函数都具有引用透明性，这个类会更容易测试，其行为也更容易理解。

JavaScript的类模型有一个问题：无法强制要求类的不可变性——始终有办法修改对象的内部数据。尽管如此，在编写一个类的时候明确说明并鼓励不可变性，通常也就足够了。尽量让类保持不可变通常是一个好的策略，以参数取代查询则是达成这一策略的利器。

11.7　移除设值函数（Remove Setting Method）



```
class Person {
  get name() {...}
  set name(aString) {...}
```

```
class Person {
  get name() {...}
```

动机

如果为某个字段提供了设值函数，这就暗示这个字段可以被改变。如果不希望在对象创建之后此字段还有机会被改变，那就不要为它提供设值函数（同时将该字段声明为不可变）。这样一来，该字段就只能在构造函数中赋值，我“不想让它被修改”的意图会更加清晰，并且可以排除其值被修改的可能性——这种可能性往往是非常大的。

有两种常见的情况需要讨论。一种情况是，有些人喜欢始终通过访问函数来读写字段值，包括在构造函数内也是如此。这会导致构造函数成为设值函数的唯一使用者。若果真如此，我更愿意去除设

值函数，清晰地表达“构造之后不应该再更新字段值”的意图。

另一种情况是，对象是由客户端通过创建脚本构造出来，而不是只有一次简单的构造函数调用。所谓“创建脚本”，首先是调用构造函数，然后就是一系列设值函数的调用，共同完成新对象的构造。创建脚本执行完以后，这个新生对象的部分（乃至全部）字段就不应该再被修改。设值函数只应该在起初的对象创建过程中调用。对于这种情况，我也会想办法去除设值函数，更清晰地表达我的意图。

做法

- 如果构造函数尚无法得到想要设入字段的值，就使用改变函数声明（124）将这个值以参数的形式传入构造函数。在构造函数中调用设值函数，对字段设值。

  如果想移除多个设值函数，可以一次性把它们的值都传入构造函数，这能简化后续步骤。

- 移除所有在构造函数之外对设值函数的调用，改为使用新的构造函数。每次修改之后都要测试。

如果不能把“调用设值函数”替换为“创建一个新对象”（例如你需要更新一个多处共享引用的对象），请放弃本重构。

- 使用内联函数（115）消去设值函数。如果可能的话，把字段声明为不可变。
- 测试。

范例

我有一个很简单的Person类。

class Person...

```
get name() {return this._name;}
set name(arg) {this._name = arg;}
get id() {return this._id;}
```

```
set id(arg) {this._id = arg;}
```

目前我会这样创建新对象：

```
const martin = new Person();
martin.name = "martin";
martin.id = "1234";
```

对象创建之后，name字段可能会改变，但id字段不会。为了更清晰地表达这个设计意图，我希望移除对应id字段的设值函数。

但id字段还得设置初始值，所以我首先用改变函数声明（124）在构造函数中添加对应的参数。

class Person...

```
constructor(id) {
  this.id = id;
}
```

然后调整创建脚本，改为从构造函数设值id字段值。

```
const martin = new Person("1234");
martin.name = "martin";
martin.id = "1234";
```

所有创建Person对象的地方都要如此修改，每次修改之后要执行测试。

全部修改完成后，就可以用内联函数（115）消去设值函数。

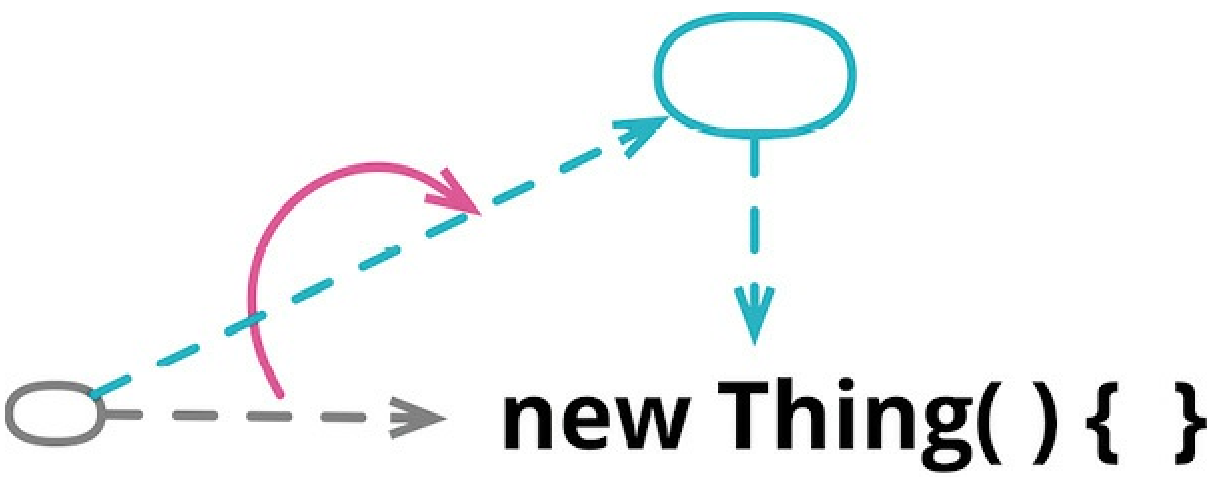
class Person...

```
constructor(id) {
  this._id = id;
}
get name() {return this._name;}
set name(arg) {this._name = arg;}
get id() {return this._id;}
```

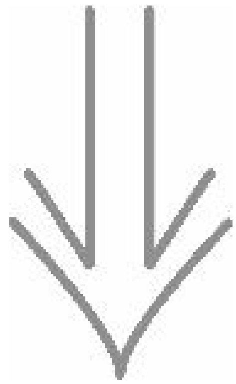
~~set id(arg) {this._id = arg;}~~

11.8　以工厂函数取代构造函数（Replace Constructor with Factory Function）

曾用名：以工厂函数取代构造函数（Replace Constructor with Factory Method）




```
leadEngineer = new Employee(document.leadEngineer, 'E');
```



```
leadEngineer = createEngineer(document.leadEngineer);
```

动机

很多面向对象语言都有特别的构造函数，专门用于对象的初始化。需要新建一个对象时，客户端通常会调用构造函数。但与一般的函数相比，构造函数又常有一些丑陋的局限性。例如，Java的构造函数只能返回当前所调用类的实例，也就是说，我无法根据环境或参数信息返回子类实例或代理对象；构造函数的名字是固定的，因此无法使用比默认名字更清晰的函数名；构造函数需要通过特殊的操作符来调用（在很多语言中是new关键字），所以在要求普通函数的场合就难以使用。

工厂函数就不受这些限制。工厂函数的实现内部可以调用构造函数，但也可以换成别的方式实

现。

做法

- 新建一个工厂函数，让它调用现有的构造函数。
- 将调用构造函数的代码改为调用工厂函数。
- 每修改一处，就执行测试。
- 尽量缩小构造函数的可见范围。

范例

又是那个单调乏味的例子：员工薪资系统。我还是以Employee类表示“员工”。

class Employee...

```
constructor (name, typeCode) {
  this._name = name;
  this._typeCode = typeCode;
}
get name() {return this._name;}
get type() {
  return Employee.legalTypeCodes[this._typeCode];
```

```
}
static get legalTypeCodes() {
  return {"E": "Engineer", "M": "Manager", "S": "Salesman"};
}
```

使用它的代码有这样的：

调用方...

```
candidate = new Employee(document.name, document.empType);
```

也有这样的：

调用方...

```
const leadEngineer = new Employee(document.leadEngineer, 'E')
```

重构的第一步是创建工厂函数，其中把对象创建的责任直接委派给构造函数。

顶层作用域...

```
function createEmployee(name, typeCode) {
  return new Employee(name, typeCode);
```

```
}
```

然后找到构造函数的调用者，并逐一修改它们，令其使用工厂函数。

第一处的修改很简单。

调用方...

```
candidate = createEmployee(document.name, document.empType);
```

第二处则可以这样使用工厂函数。

调用方...

```
const leadEngineer = createEmployee(document.leadEngineer, 'E
```

但我不喜欢这里的类型码——以字符串字面量的形式传入类型码，一般来说都是坏味道。所以我更愿意再新建一个工厂函数，把“员工类别”的信息嵌在函数名里体现。

调用方...

```
const leadEngineer = createEngineer(document.leadEngineer);
```

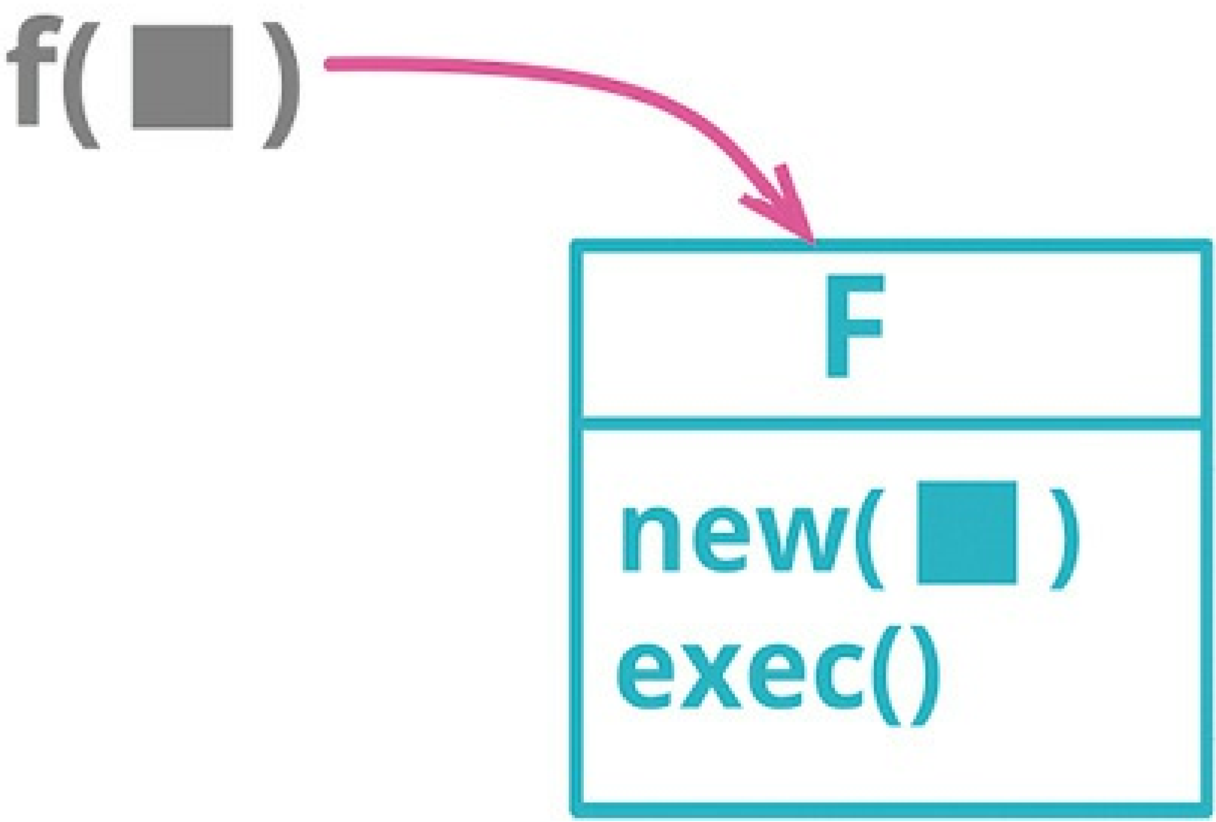
顶层作用域...

```
function createEngineer(name) {
  return new Employee(name, 'E');
}
```

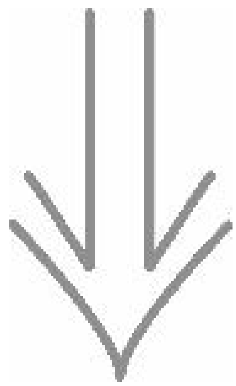
11.9　以命令取代函数（Replace Function with Command）

曾用名：以函数对象取代函数（Replace Method with Method Object）

反向重构：以函数取代命令（344）



```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  // long body code
}
```



```
class Scorer {
  constructor(candidate, medicalExam, scoringGuide) {
    this._candidate = candidate;
    this._medicalExam = medicalExam;
    this._scoringGuide = scoringGuide;
  }

  execute() {
    this._result = 0;
    this._healthLevel = 0;
    // long body code
  }
}
```

动机

函数，不管是独立函数，还是以方法

（method）形式附着在对象上的函数，是程序设计的基本构造块。不过，将函数封装成自己的对象，有时也是一种有用的办法。这样的对象我称之为“命令对象”（command object），或者简称“命令”（command）。这种对象大多只服务于单一函数，获得对该函数的请求，执行该函数，就是这种对象存在的意义。

与普通的函数相比，命令对象提供了更大的控制灵活性和更强的表达能力。除了函数调用本身，命令对象还可以支持附加的操作，例如撤销操作。我可以通过命令对象提供的方法来设值命令的参数值，从而支持更丰富的生命周期管理能力。我可以借助继承和钩子对函数行为加以定制。如果我所使用的编程语言支持对象但不支持函数作为一等公民，通过命令对象就可以给函数提供大部分相当于一等公民的能力。同样，即便编程语言本身并不支持嵌套函数，我也可以借助命令对象的方法和字段把复杂的函数拆解开，而且在测试和调试过程中可以直接调用这些方法。

所有这些都是使用命令对象的好理由，所以我要做好准备，一旦有需要，就能把函数重构成命令。不过我们不能忘记，命令对象的灵活性也是以复杂性作为代价的。所以，如果要在作为一等公民的函数和命令对象之间做个选择，95%的时候我都

会选函数。只有当我特别需要命令对象提供的某种能力而普通的函数无法提供这种能力时，我才会考虑使用命令对象。

> 跟软件开发中的很多词汇一样，“命令”这个词承载了太多含义。在这里，“命令”是指一个对象，其中封装了一个函数调用请求。这是遵循《设计模式》[gof]一书中的命令模式（command pattern）。在这个意义上，使用“命令”一词时，我会先用完整的“命令对象”一词设定上下文，然后视情况使用简略的“命令”一词。在命令与查询分离原则（command-query separation principle）中也用到了“命令”一词，此时“命令”是一个对象所拥有的函数，调用该函数可以改变对象可观察的状态。我尽量避免使用这个意义上的“命令”一词，而更愿意称其为“修改函数”（modifier）或者“改变函数”（mutator）。

做法

- 为想要包装的函数创建一个空的类，根据该函数的名字为其命名。
- 使用搬移函数（198）把函数移到空的类里。

保持原来的函数作为转发函数，至少保留到重构结束之前才删除。

遵循编程语言的命名规范来给命令对象起名。如果没有合适的命名规范，就给命令对象中负责实际执行命令的函数起一个通用的名字，例如“`execute`”或者“`call`”。

- 可以考虑给每个参数创建一个字段，并在构造函数中添加对应的参数。

范例

JavaScript语言有很多缺点，但把函数作为一等公民对待，是它最正确的设计决策之一。在不具备这种能力的编程语言中，我经常要费力为很常见的任务创建命令对象，JavaScript则省去了这些麻烦。不过，即便在JavaScript中，有时也需要用到命

令对象。

一个典型的应用场景就是拆解复杂的函数，以便我理解和修改。要想真正展示这个重构手法的价值，我需要一个长而复杂的函数，但这写起来太费事，你读起来也麻烦。所以我在这里展示的函数其实很短，并不真的需要本重构手法，还望读者权且包涵。下面的函数用于给一份保险申请评分。

```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";

if (scoringGuide.stateWithLowCertification(candidate.originSt
    certificationGrade = "low";
    result -= 5;
  }
  // lots more code like this
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

我首先创建一个空的类，用搬移函数（198）把上述函数搬到这个类里去。

```
function score(candidate, medicalExam, scoringGuide) {

return new Scorer().execute(candidate, medicalExam, scoringGu
```

```
}

  class Scorer {
    execute (candidate, medicalExam, scoringGuide) {
      let result = 0;
      let healthLevel = 0;
      let highMedicalRiskFlag = false;

      if (medicalExam.isSmoker) {
        healthLevel += 10;
        highMedicalRiskFlag = true;
      }
    let certificationGrade = "regular";

if (scoringGuide.stateWithLowCertification(candidate.originSt
      certificationGrade = "low";
      result -= 5;
    }
    // lots more code like this
    result -= Math.max(healthLevel - 5, 0);
    return result;
  }
}
```

大多数时候，我更愿意在命令对象的构造函数中传入参数，而不让execute函数接收参数。在这样一个简单的拆解场景中，这一点带来的影响不大；但如果我要处理的命令需要更复杂的参数设置周期或者大量定制，上述做法就会带来很多便利：多个命令类可以分别从各自的构造函数中获得各自不同的参数，然后又可以排成队列挨个执行，因为它们的execute函数签名都一样。

我可以每次搬移一个参数到构造函数。

```
function score(candidate, medicalExam, scoringGuide) {
```

```
  return new Scorer(candidate).execute(candidate, medicalExam
}
```

class Scorer...

```
constructor(candidate){
  this._candidate = candidate;
}

execute (candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";

if (scoringGuide.stateWithLowCertification(this._candidate.or
    certificationGrade = "low";
    result -= 5;
  }
  // lots more code like this
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

继续处理其他参数：

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer(candidate, medicalExam, scoringGuide).exe
}
```

class Scorer...

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}
execute () {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";

if (this._scoringGuide.stateWithLowCertification(this._candid
    certificationGrade = "low";
    result -= 5;
  }
  // lots more code like this
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

以命令取代函数的重构到此就结束了，不过之所以要做这个重构，是为了拆解复杂的函数，所以我还是大致展示一下如何拆解。下一步是把所有局部变量都变成字段，我还是每次修改一处。

class Scorer...

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}

execute () {
  this._result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";

if (this._scoringGuide.stateWithLowCertification(this._candid
    certificationGrade = "low";
    this._result -= 5;
  }
  // lots more code like this
  this._result -= Math.max(healthLevel - 5, 0);
  return this._result;
}
```

重复上述过程，直到所有局部变量都变成字段。（“把局部变量变成字段”这个重构手法是如此简单，以至于我都没有在重构名录中给它一席之地。对此我略感愧疚。）

class Scorer...

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
```

```
  }

  execute () {
    this._result = 0;
    this._healthLevel = 0;
    this._highMedicalRiskFlag = false;

    if (this._medicalExam.isSmoker) {
      this._healthLevel += 10;
      this._highMedicalRiskFlag = true;
    }
    this._certificationGrade = "regular";

  if (this._scoringGuide.stateWithLowCertification(this._candid
      this._certificationGrade = "low";
      this._result -= 5;
    }
    // lots more code like this
    this._result -= Math.max(this._healthLevel - 5, 0);
    return this._result;
  }
```

现在函数的所有状态都已经移到了命令对象中，我可以放心使用提炼函数（106）等重构手法，而不用纠结于局部变量的作用域之类问题。

class Scorer...

```
  execute () {
    this._result = 0;
    this._healthLevel = 0;
    this._highMedicalRiskFlag = false;

    this.scoreSmoking();
    this._certificationGrade = "regular";

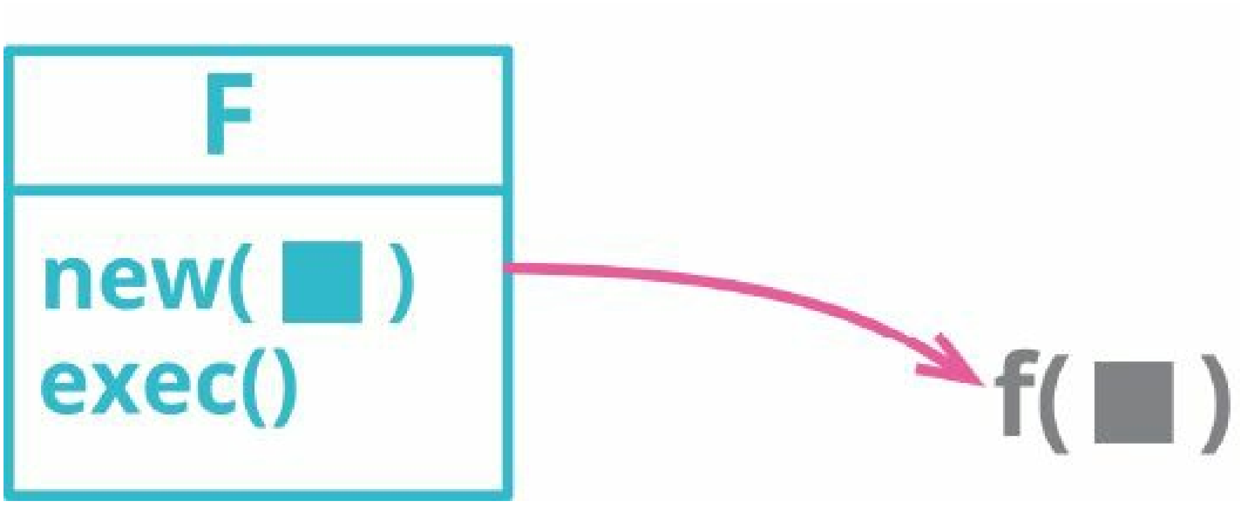
  if (this._scoringGuide.stateWithLowCertification(this._candid
      this._certificationGrade = "low";
```

```
      this._result -= 5;
    }
    // lots more code like this
    this._result -= Math.max(this._healthLevel - 5, 0);
    return this._result;
    }
  scoreSmoking() {
    if (this._medicalExam.isSmoker) {
      this._healthLevel += 10;
      this._highMedicalRiskFlag = true;
    }
  }
```

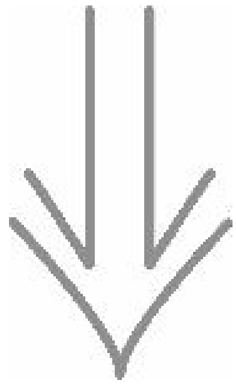
这样我就可以像处理嵌套函数一样处理命令对象。实际上，在JavaScript中运用此重构手法时，的确可以考虑用嵌套函数来代替命令对象。不过我还是会使用命令对象，不仅因为我对命令对象更熟悉，而且还因为我可以针对命令对象中任何一个函数进行测试和调试。

11.10　以函数取代命令（Replace Command with Function）

反向重构：以命令取代函数（337）




```
class ChargeCalculator {
  constructor (customer, usage){
    this._customer = customer;
    this._usage = usage;
  }
  execute() {
    return this._customer.rate * this._usage;
  }
}
```



```
function charge(customer, usage) {
  return customer.rate * usage;
}
```

动机

命令对象为处理复杂计算提供了强大的机制。借助命令对象，可以轻松地将原本复杂的函数拆解为多个方法，彼此之间通过字段共享状态；拆解后的方法可以分别调用；开始调用之前的数据状态也可以逐步构建。但这种强大是有代价的。大多数时候，我只是想调用一个函数，让它完成自己的工作就好。如果这个函数不是太复杂，那么命令对象可能显得费而不惠，我就应该考虑将其变回普通的函数。

做法

- 运用提炼函数（106），把“创建并执行命令对象”的代码单独提炼到一个函数中。

  这一步会新建一个函数，最终这个函数会取代现在的命令对象。

- 对命令对象在执行阶段用到的函数，逐一使用内联函数（115）。

  如果被调用的函数有返回值，请先对调用处使用提炼变量（119），然后再使用内联函数（115）。

- 使用改变函数声明（124），把构造函数的参数转移到执行函数。
- 对于所有的字段，在执行函数中找到引用它们的地方，并改为使用参数。每次修改后都要测试。

- 把“调用构造函数”和“调用执行函数”两步都内联到调用方（也就是最终要替换命令对象的那个函数）。
- 测试。
- 用移除死代码（237）把命令类消去。

范例

假设我有一个很小的命令对象。

```
class ChargeCalculator {
  constructor (customer, usage, provider){
    this._customer = customer;
    this._usage = usage;
    this._provider = provider;
  }
  get baseCharge() {
    return this._customer.baseRate * this._usage;
  }
  get charge() {
    return this.baseCharge + this._provider.connectionCharge;
  }
}
```

使用方的代码如下。

调用方...

```
monthCharge = new ChargeCalculator(customer, usage, provider)
```

命令类足够小、足够简单，变成函数更合适。

首先，我用提炼函数（106）把命令对象的创建与调用过程包装到一个函数中。

调用方**...**

```
monthCharge = charge(customer, usage, provider);
```

顶层作用域**...**

```
function charge(customer, usage, provider) {
  return new ChargeCalculator(customer, usage, provider).char
}
```

接下来要考虑如何处理支持函数（也就是这里的`baseCharge`函数）。对于有返回值的函数，我一般会先用提炼变量（119）把返回值提炼出来。

class ChargeCalculator...

```
get baseCharge() {
  return this._customer.baseRate * this._usage;
}
get charge() {
  const baseCharge = this.baseCharge;
  return baseCharge + this._provider.connectionCharge;
}
```

然后对支持函数使用内联函数（115）。

class ChargeCalculator...

```
get charge() {
  const baseCharge = this._customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

现在所有逻辑处理都集中到一个函数了，下一步是把构造函数传入的数据移到主函数。首先用改变函数声明（124）把构造函数的参数逐一添加到charge函数上。

class ChargeCalculator...

```
constructor (customer, usage, provider){
  this._customer = customer;
  this._usage = usage;
  this._provider = provider;
}
```

```
charge(customer, usage, provider) {
  const baseCharge = this._customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

顶层作用域...

```
function charge(customer, usage, provider) {
  return new ChargeCalculator(customer, usage, provider)
                      .charge(customer, usage, provider);
}
```

然后修改charge函数的实现，改为使用传入的参数。这个修改可以小步进行，每次使用一个参数。

class ChargeCalculator...

```
constructor (customer, usage, provider){
  this._customer = customer;
  this._usage = usage;
  this._provider = provider;
}

charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

构造函数中对this._customer字段的赋值不删除也没关系，因为反正没人使用这个字段。但我更愿意去掉这条赋值语句，因为去掉它以后，如果在函数实现中漏掉了一处对字段的使用没有修改，测试就会失败。（如果我真的犯了这个错误而测试没有失败，我就应该考虑增加测试了。）

其他参数也如法炮制，直到charge函数不再使用任何字段：

class ChargeCalculator...

```
charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * usage;
  return baseCharge + provider.connectionCharge;
}
```

现在我就可以把所有逻辑都内联到顶层的charge函数中。这是内联函数（115）的一种特殊情况，我需要把构造函数和执行函数一并内联。

顶层作用域...

```
function charge(customer, usage, provider) {
    const baseCharge = customer.baseRate * usage;
    return baseCharge + provider.connectionCharge;
}
```

现在命令类已经是死代码了，可以用移除死代码（237）给它一个体面的葬礼。

第12章　处理继承关系

在最后一章里，我将介绍面向对象编程技术里最为人熟知的一个特性：继承。与任何强有力的特性一样，继承机制十分实用，却也经常被误用，而且常得等你用上一段时间，遇见了痛点，才能察觉误用所在。

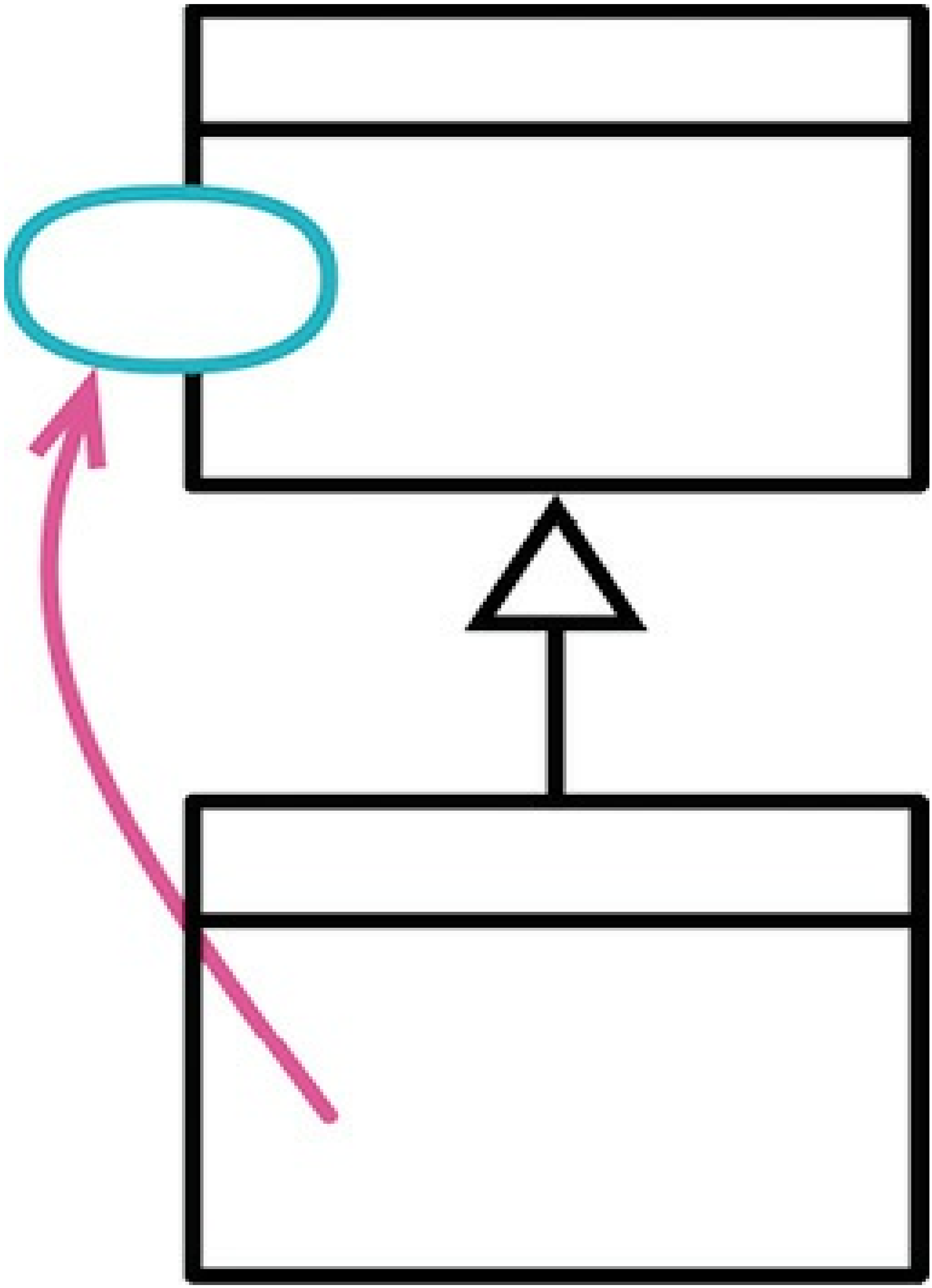
特性（主要是函数和字段）经常需要在继承体系里上下调整。我有一组手法专门用来处理此类调整：函数上移（350）、字段上移（353）、构造函数本体上移（355）、函数下移（359）以及字段下移（361）。我可以使用提炼超类（375）、移除子类（369）以及折叠继承体系（380）来为继承体系添加新类或删除旧类。如果一个字段仅仅作为类型码使用，根据其值来触发不同的行为，那么我会通过以子类取代类型码（362），用一个子类来取代这样的字段。

继承本身是一个强有力的工具，但有时它也可能被用于错误的地方，有时本来适合使用继承的场景变得不再合适——若果真如此，我就会用以委托取代子类（381）或以委托取代超类（399）将继承

体系转化成委托调用。

12.1 函数上移（Pull Up Method）

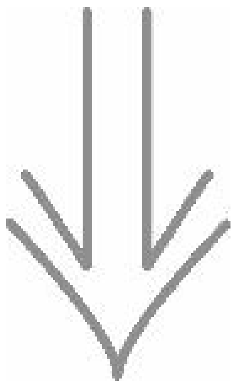
反向重构：函数下移（359）



```
class Employee {...}

class Salesman extends Employee {
  get name() {...}
}

class Engineer extends Employee {
  get name() {...}
}
```



```
class Employee {
  get name() {...}
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

动机

避免重复代码是很重要的。重复的两个函数现在也许能够正常工作，但假以时日却只会成为滋生

bug的温床。无论何时，只要系统内出现重复，你就会面临“修改其中一个却未能修改另一个”的风险。通常，找出重复也有一定的难度。

如果某个函数在各个子类中的函数体都相同（它们很可能是通过复制粘贴得到的），这就是最显而易见的函数上移适用场合。当然，情况并不总是如此明显。我也可以只管放心地重构，再看看测试程序会不会发牢骚，但这就需要对我的测试有充分的信心。我发现，观察这些可能重复的函数之间的差异往往大有收获：它们经常会向我展示那些我忘记测试的行为。

函数上移常常紧随其他重构而被使用。也许我能找出若干个身处不同子类内的函数，而它们又可以通过某种形式的参数调整成为相同的函数。这时候，最简单的办法就是先分别对这些函数应用函数参数化（310），然后应用函数上移。

函数上移过程中最麻烦的一点就是，被提升的函数可能会引用只出现于子类而不出现于超类的特性。此时，我就得用字段上移（353）和函数上移先将这些特性（类或者函数）提升到超类。

如果两个函数工作流程大体相似，但实现细节略有差异，那么我会考虑先借助塑造模板函数（Form Template Method）[mf-ft]构造出相同的函

数，然后再提升它们。

做法

- 检查待提升函数，确定它们是完全一致的。

　　如果它们做了相同的事情，但函数体并不完全一致，那就先对它们进行重构，直到其函数体完全一致。

- 检查函数体内引用的所有函数调用和字段都能从超类中调用到。
- 如果待提升函数的签名不同，使用改变函数声明（124）将那些签名都修改为你想要在超类中使用的签名。
- 在超类中新建一个函数，将某一个待提升函数的代码复制到其中。
- 执行静态检查。
- 移除一个待提升的子类函数。
- 测试。
- 逐一移除待提升的子类函数，直到只剩下超类

中的函数为止。

范例

我手上有两个子类，它们之中各有一个函数做了相同的事情：

class Employee extends Party...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

class Department extends Party...

```
get totalAnnualCost() {
  return this.monthlyCost * 12;
}
```

检查两个类的函数时我发现，两个函数都引用了`monthlyCost`属性，但后者并未在超类中定义，而是在两个子类中各自定义了一份实现。因为

JavaScript是动态语言，这样做没有问题；但如果是在一门静态语言里，我就必须将monthlyCost声明为Party类上的抽象函数，否则编译器就会报错。

两个函数各有不同的名字，因此第一步是用改变函数声明（124）统一它们的函数名。

class Department...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

然后，我从其中一个子类中将annualCost函数复制到超类。

class Party...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

在静态语言里，做完这一步我就可以编译一次，确保超类函数的所有引用都能正常工作。但这是在JavaScript里，编译显然帮不上什么忙，因此我

直接先从Employee中移除annualCost函数，测试，接着移除Department类中的annualCost函数。

这项重构手法至此即告完成，但还有一个遗留问题需要解决：annualCost函数中调用了monthlyCost，但后者并未在Party类中显式声明。当然代码仍能正常工作，这得益于JavaScript是动态语言，它能自动帮你调用子类上的同名函数。但若能明确传达出“继承Party类的子类需要提供一个monthlyCost实现”这个信息，无疑也有很大的价值，特别是对日后需要添加子类的后来者。其中一种好的传达方式是添加一个如下的陷阱（trap）函数。

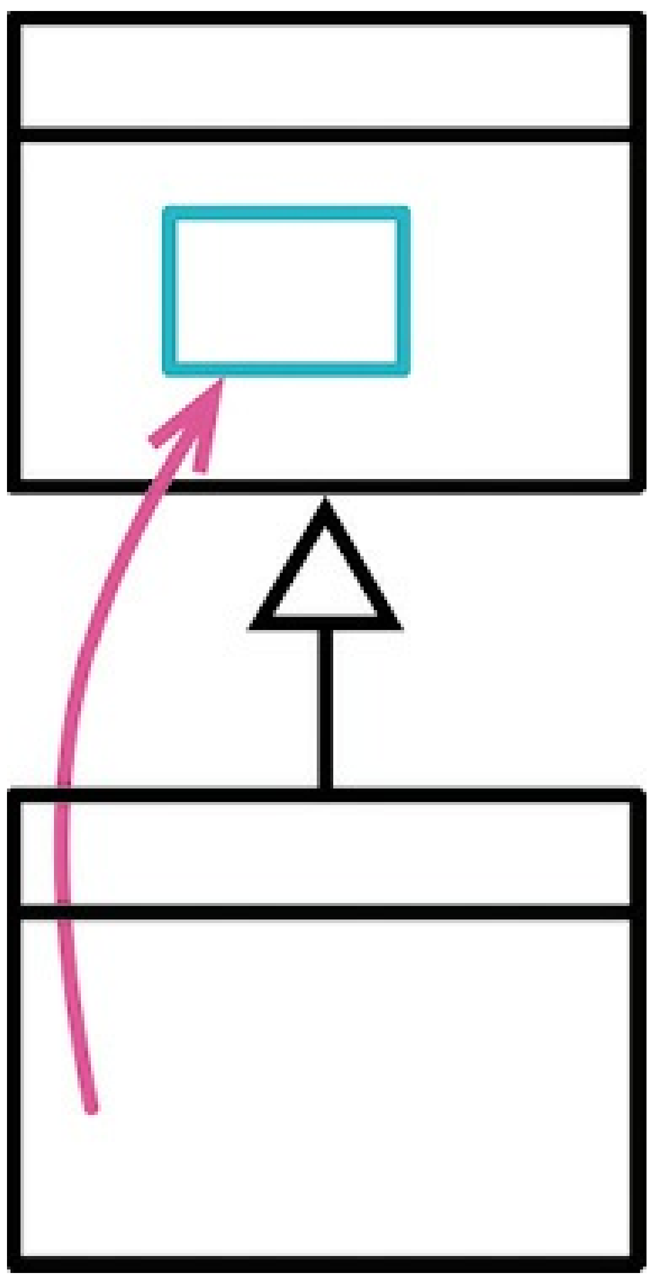
class Party...

```
get monthlyCost() {
  throw new SubclassResponsibilityError();
}
```

我称上述抛出的错误为一个“子类未履行职责错误”，这是从Smalltalk借鉴来的名字。

12.2 字段上移（Pull Up Field）

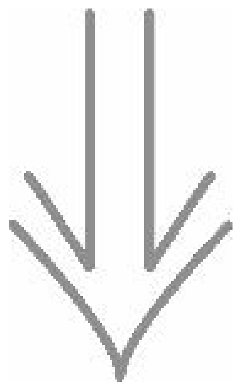
反向重构：字段下移（361）



```java
class Employee {...} // Java

class Salesman extends Employee {
  private String name;
}

class Engineer extends Employee {
  private String name;
}
```



```java
class Employee {
  protected String name;
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

动机

如果各子类是分别开发的，或者是在重构过程中组合起来的，你常会发现它们拥有重复特性，特别是字段更容易重复。这样的字段有时拥有近似的

名字，但也并非绝对如此。判断若干字段是否重复，唯一的办法就是观察函数如何使用它们。如果它们被使用的方式很相似，我就可以将它们提升到超类中去。

本项重构可从两方面减少重复：首先它去除了重复的数据声明；其次它使我可以将使用该字段的行为从子类移至超类，从而去除重复的行为。

许多动态语言不需要在类定义中定义字段，相反，字段是在第一次被赋值的同时完成声明。在这种情况下，字段上移基本上是应用构造函数本体上移（355）后的必然结果。

做法

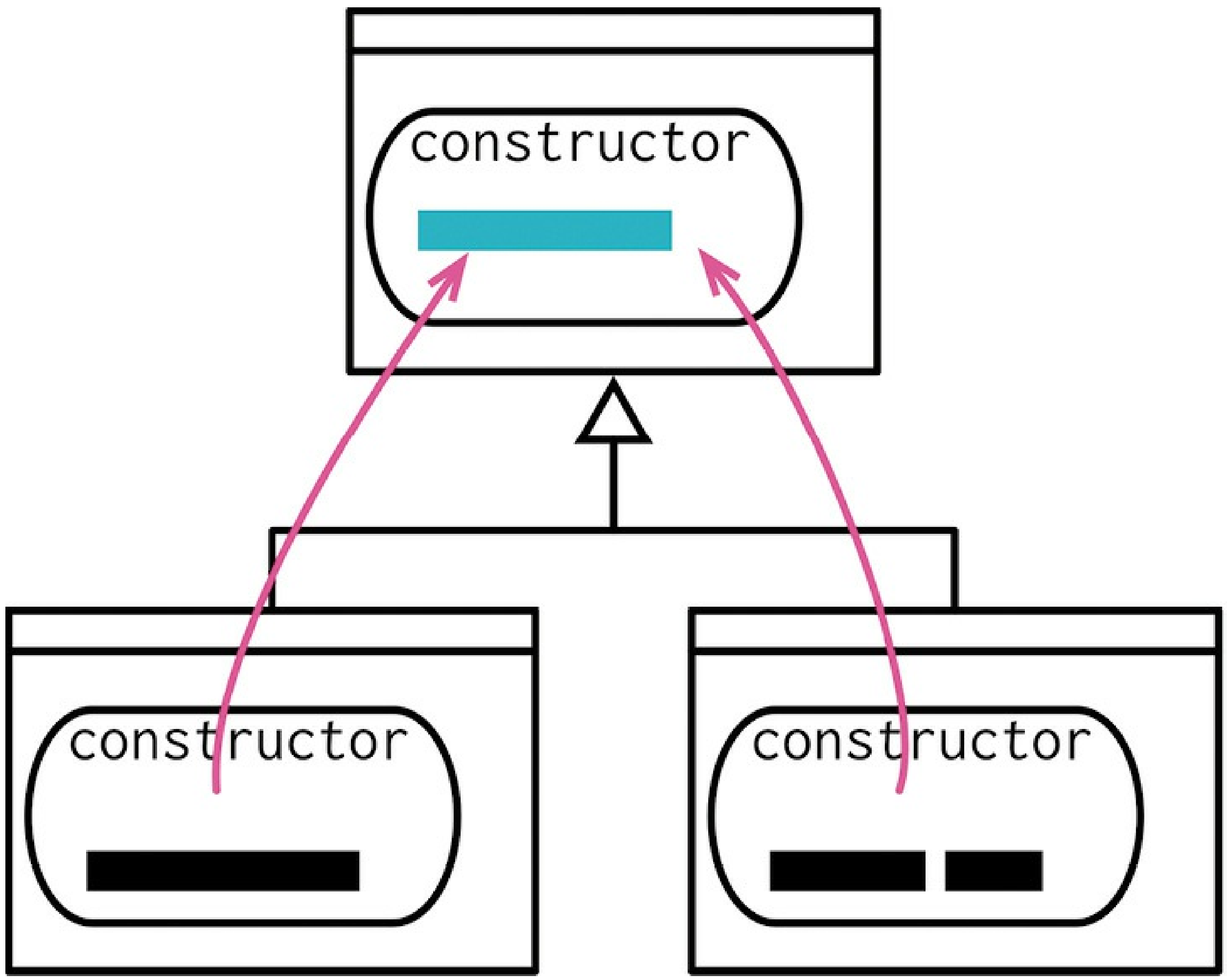
- 针对待提升之字段，检查它们的所有使用点，确认它们以同样的方式被使用。
- 如果这些字段的名称不同，先使用变量改名（137）为它们取个相同的名字。
- 在超类中新建一个字段。

新字段需要对所有子类可见（在大多数语

言中protected权限便已足够）。

- 移除子类中的字段。
- 测试。

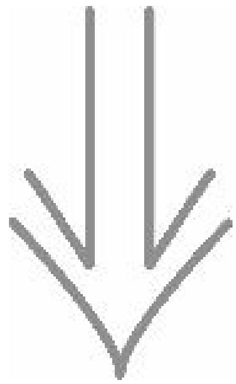
12.3　构造函数本体上移（Pull Up Constructor Body）




```
class Party {...}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
```

```
    this._monthlyCost = monthlyCost;
  }
}
```



```
class Party {
  constructor(name){
    this._name = name;
  }
}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super(name);
    this._id = id;
    this._monthlyCost = monthlyCost;
  }
}
```

动机

构造函数是很奇妙的东西。它们不是普通函数，使用它们比使用普通函数受到更多的限制。

如果我看见各个子类中的函数有共同行为，我的第一个念头就是使用提炼函数（106）将它们提炼到一个独立函数中，然后使用函数上移（350）将这个函数提升至超类。但构造函数的出现打乱了我的算盘，因为它们附加了特殊的规则，对一些做法与函数的调用次序有所限制。要对付它们，我需要略微不同的做法。

如果重构过程过于复杂，我会考虑转而使用以工厂函数取代构造函数（334）。

做法

- 如果超类还不存在构造函数，首先为其定义一个。确保让子类调用超类的构造函数。
- 使用移动语句（223）将子类中构造函数中的公共语句移动到超类的构造函数调用语句之后。
- 逐一移除子类间的公共代码，将其提升至超类构造函数中。对于公共代码中引用到的变量，将其作为参数传递给超类的构造函数。
- 测试。
- 如果存在无法简单提升至超类的公共代码，先应用提炼函数（106），再利用函数上移（350）提升之。

范例

我以下列“雇员”的例子开始：

```
class Party {}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // rest of class...
class Department extends Party {
  constructor(name, staff){
    super();
    this._name = name;
    this._staff = staff;
  }
  // rest of class...
```

Party的两个子类间存在公共代码，也即是对名字（name）的赋值。我先用移动语句（223）将Employee中的这行赋值语句移动到super()调用后面：

```
class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._name = name;
    this._id = id;
    this._monthlyCost = monthlyCost;
  }
```

```
  // rest of class...
```

测试。之后我将这行公共代码提升至超类的构造函数中。由于其中引用了一个子类构造函数传入的参数name，于是我将该参数一并传给超类构造函数。

class Party...

```
constructor(name){
  this._name = name;
}
```

class Employee...

```
constructor(name, id, monthlyCost) {
  super(name);
  this._id = id;
  this._monthlyCost = monthlyCost;
}
```

class Department...

```
constructor(name, staff){
```

```
    super(name);
    this._staff = staff;
  }
```

运行测试。然后大功告成。

多数时候，一个构造函数的工作原理都是这样：先（通过super调用）初始化共用的数据，再由各个子类完成额外的工作。但是，偶尔也需要将共用行为的初始化提升至超类，这时问题便来了。

请看下面的例子。

class Employee...

```
  constructor (name) {...}

  get isPrivileged() {...}

  assignCar() {...}
```

class Manager extends Employee...

```
  constructor(name, grade) {
    super(name);
    this._grade = grade;
    if (this.isPrivileged) this.assignCar(); // every subclass
  }
```

```
get isPrivileged() {
  return this._grade >4;
}
```

这里我无法简单地提升isPrivileged函数至超类，因为调用它之前需要先为grade字段赋值，而该字段只能在子类的构造函数中初始化。

在这种场景下，我可以对这部分公共代码使用提炼函数（106）。

class Manager...

```
constructor(name, grade) {
  super(name);
  this._grade = grade;
  this.finishConstruction();
}

finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

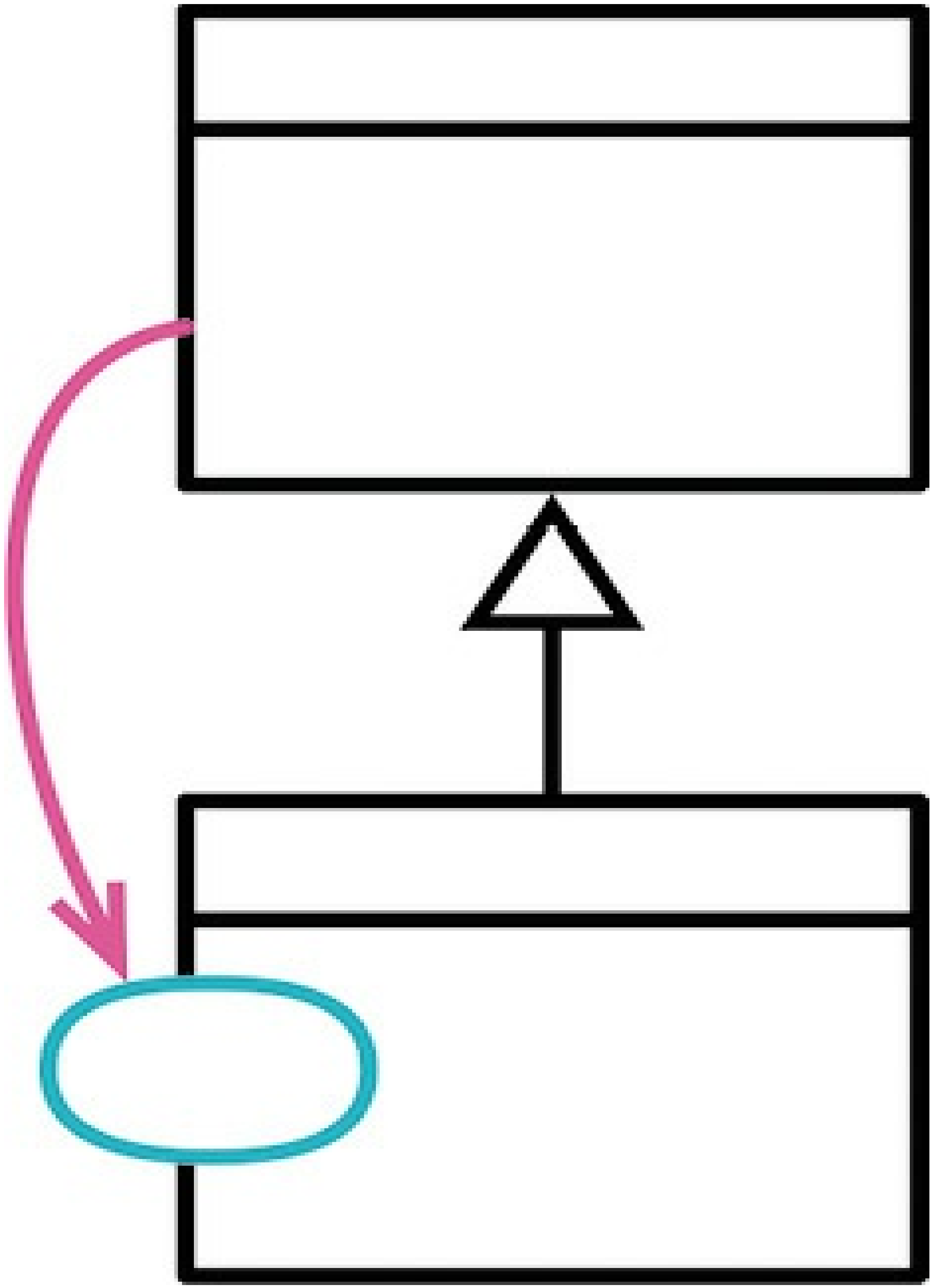
然后再使用函数上移（350）将提炼得到的函数提升至超类。

class Employee...

```
finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

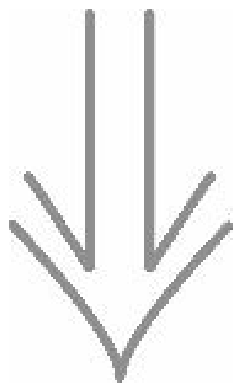
12.4 函数下移（Push Down Method）

反向重构：函数上移（350）



```
class Employee {
  get quota {...}
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```



```
class Employee {...}
class Engineer extends Employee {...}
class Salesman extends Employee {
  get quota {...}
}
```

动机

如果超类中的某个函数只与一个（或少数几个）子类有关，那么最好将其从超类中挪走，放到真正关心它的子类中去。这项重构手法只有在超类明确知道哪些子类需要这个函数时适用。如果超类

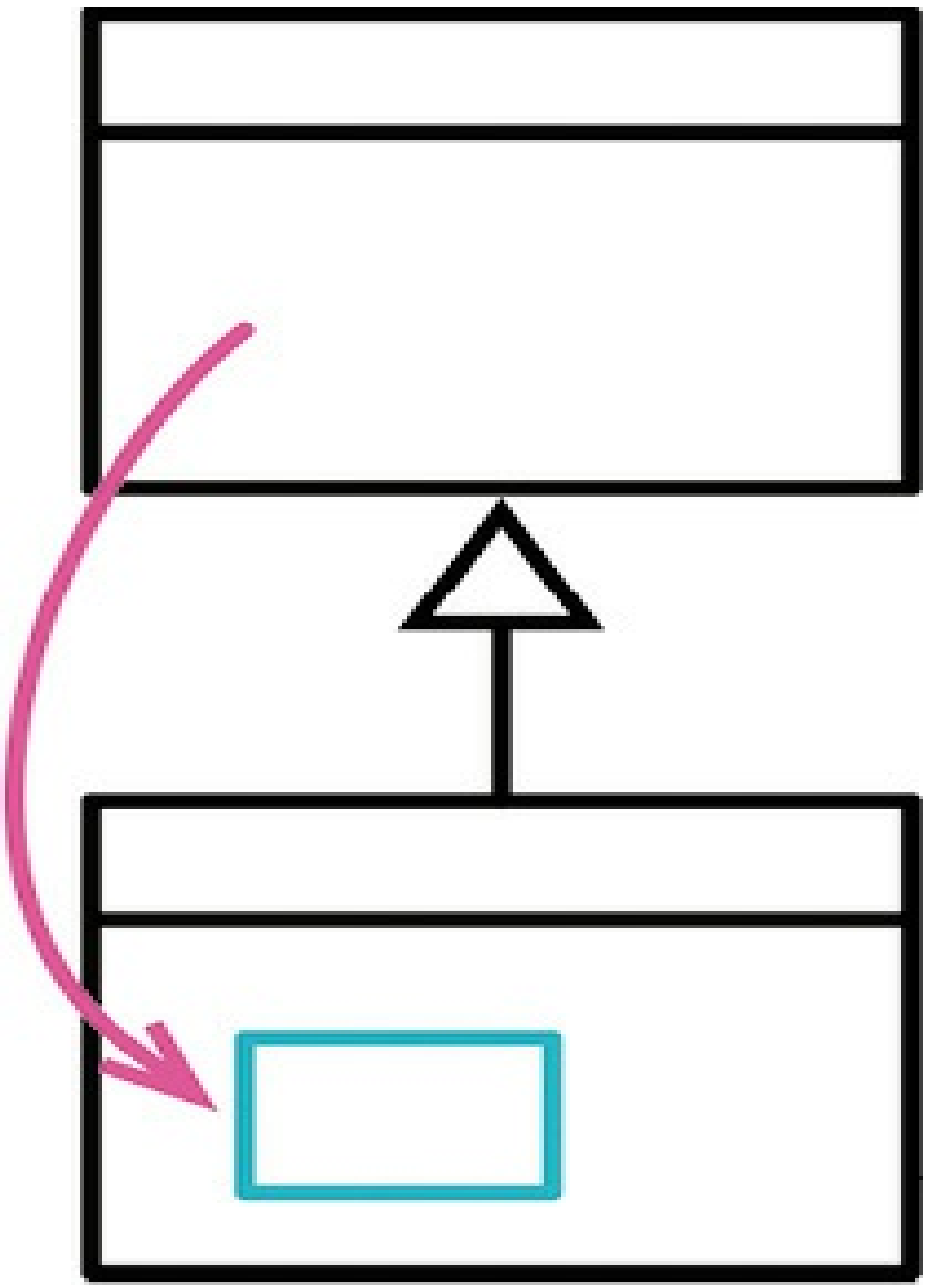
不知晓这个信息，那我就得用以多态取代条件表达式（272），只留些共用的行为在超类。

做法

- 将超类中的函数本体复制到每一个需要此函数的子类中。
- 删除超类中的函数。
- 测试。
- 将该函数从所有不需要它的那些子类中删除。
- 测试。

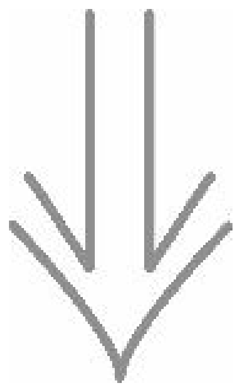
12.5　字段下移（Push Down Field）

反向重构：字段上移（353）



```
class Employee {   // Java
  private String quota;
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```



```
class Employee {...}
class Engineer extends Employee {...}

class Salesman extends Employee {
  protected String quota;
}
```

动机

如果某个字段只被一个子类（或者一小部分子类）用到，就将其搬移到需要该字段的子类中。

做法

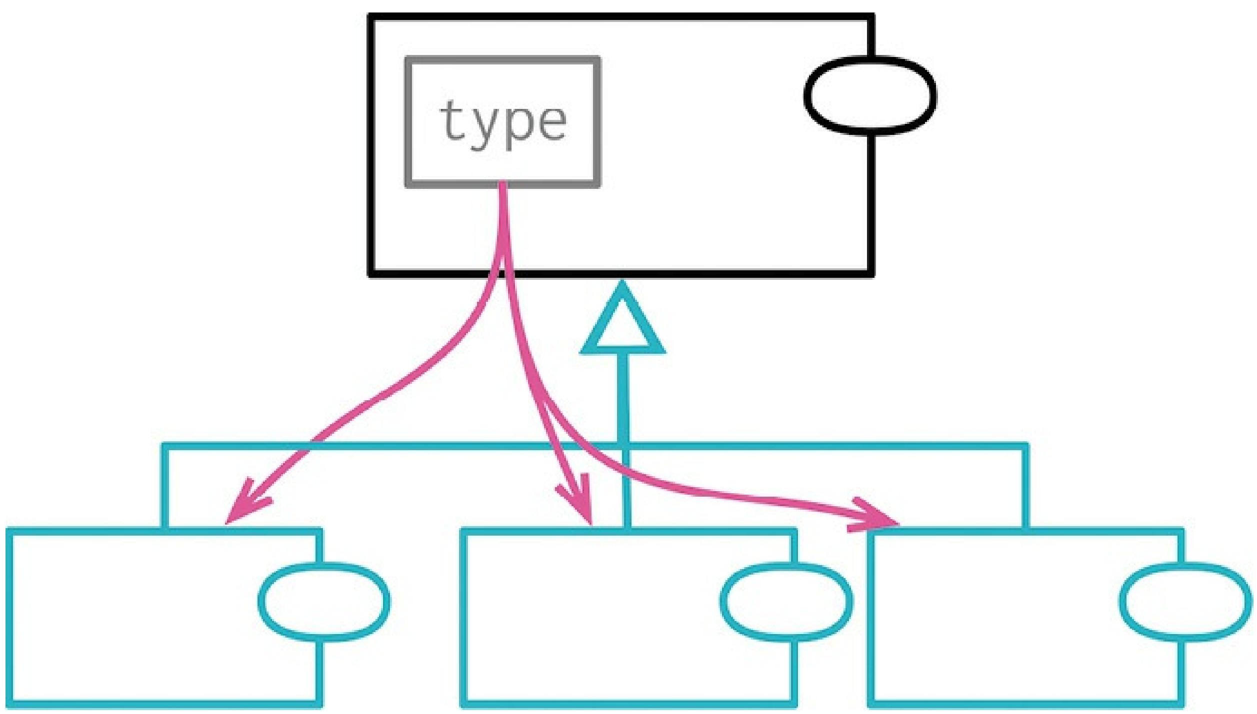
- 在所有需要该字段的子类中声明该字段。
- 将该字段从超类中移除。
- 测试。
- 将该字段从所有不需要它的那些子类中删掉。
- 测试。

12.6　以子类取代类型码（Replace Type Code with Subclasses）

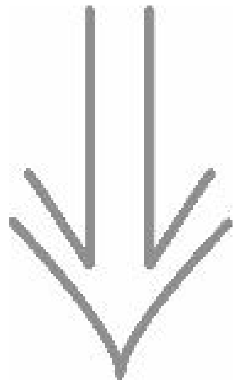
包含旧重构：以State/Strategy取代类型码（Replace Type Code with State/Strategy）

包含旧重构：提炼子类（Extract Subclass）

反向重构：移除子类（369）




```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```



```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name);
    case "salesman": return new Salesman(name);
    case "manager": return new Manager (name);
}
```

动机

软件系统经常需要表现“相似但又不同的东西”，比如员工可以按职位分类（工程师、经理、销售），订单可以按优先级分类（加急、常规）。表现分类关系的第一种工具是类型码字段——根据具体的编程语言，可能实现为枚举、符号、字符串或者数字。类型码的取值经常来自给系统提供数据的外部服务。

大多数时候，有这样的类型码就够了。但也有些时候，我可以再多往前一步，引入子类。继承有

两个诱人之处。首先，你可以用多态来处理条件逻辑。如果有几个函数都在根据类型码的取值采取不同的行为，多态就显得特别有用。引入子类之后，我可以用以多态取代条件表达式（272）来处理这些函数。

另外，有些字段或函数只对特定的类型码取值才有意义，例如“销售目标”只对“销售”这类员工才有意义。此时我可以创建子类，然后用字段下移（361）把这样的字段放到合适的子类中去。当然，我也可以加入验证逻辑，确保只有当类型码取值正确时才使用该字段，不过子类的形式能更明确地表达数据与类型之间的关系。

在使用以子类取代类型码时，我需要考虑一个问题：应该直接处理携带类型码的这个类，还是应该处理类型码本身呢？以前面的例子来说，我是应该让“工程师”成为“员工”的子类，还是应该在“员工”类包含“员工类别”属性、从后者继承出“工程师”和“经理”等子类型呢？直接的子类继承（前一种方案）比较简单，但职位类别就不能用在其他场合了。另外，如果员工的类别是可变的，那么也不能使用直接继承的方案。如果想在“员工类别”之下创建子类，可以运用以对象取代基本类型（174）把类型码包装成“员工类别”类，然后对其使用以子类取代类型码（362）。

做法

- 自封装类型码字段。
- 任选一个类型码取值，为其创建一个子类。覆写类型码类的取值函数，令其返回该类型码的字面量值。
- 创建一个选择器逻辑，把类型码参数映射到新的子类。

如果选择直接继承的方案，就用以工厂函数取代构造函数（334）包装构造函数，把选择器逻辑放在工厂函数里；如果选择间接继承的方案，选择器逻辑可以保留在构造函数里。

- 测试。
- 针对每个类型码取值，重复上述“创建子类、添加选择器逻辑”的过程。每次修改后执行测试。
- 去除类型码字段。
- 测试。
- 使用函数下移（359）和以多态取代条件表达式（272）处理原本访问了类型码的函数。全部处

理完后，就可以移除类型码的访问函数。

范例

这个员工管理系统的例子已经被用烂了……

class Employee...

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}
validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}
toString() {return `${this._name} (${this._type})`;}
```

第一步是用封装变量（132）将类型码自封装起来。

class Employee...

```
get type() {return this._type;}
toString() {return `${this._name} (${this.type})`;}
```

请注意，toString函数的实现中去掉了this._type的下划线，改用新建的取值函数了。

我选择从工程师（"engineer"）这个类型码开始重构。我打算采用直接继承的方案，也就是继承Employee类。子类很简单，只要覆写类型码的取值函数，返回适当的字面量值就行了。

```
class Engineer extends Employee {
  get type() {return "engineer";}
}
```

虽然JavaScript的构造函数也可以返回其他对象，但如果把选择器逻辑放在这儿，它会与字段初始化逻辑相互纠缠，搞得一团混乱。所以我会先运用以工厂函数取代构造函数（334），新建一个工厂函数以便安放选择器逻辑。

```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```

然后我把选择器逻辑放在工厂函数中，从而开始使用新的子类。

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
```

```
  }
  return new Employee(name, type);
}
```

测试，确保一切运转正常。不过由于我的偏执，我随后会修改Engineer类中覆写的type函数，让它返回另外一个值，再次执行测试，确保会有测试失败，这样我才能肯定：新建的子类真的被用到了。然后我把type函数的返回值改回正确的状态，继续处理别的类型。我一次处理一个类型，每次修改后都执行测试。

```
class Salesman extends Employee {
  get type() {return "salesman";}
}

class Manager extends Employee {
  get type() {return "manager";}
}

function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager":  return new Manager (name, type);
  }
  return new Employee(name, type);
}
```

全部修改完成后，我就可以去掉类型码字段及其在超类中的取值函数（子类中的取值函数仍然保留）。

class Employee...

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}

get type() {return this._type;}
toString() {return `${this._name} (${this.type})`;}
```

测试，确保一切工作正常，我就可以移除验证逻辑，因为分发逻辑做的是同一回事。

class Employee...

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
}
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager":  return new Manager (name, type);

default: throw new Error(`Employee cannot be of type ${type}`
  }
  return new Employee(name, type);
}
```

现在，构造函数的类型参数已经没用了，用改

变函数声明（124）把它干掉。

class Employee...

```
constructor(name, type){
  this._name = name;
}

function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager": return new Manager (name, type);

default: throw new Error(`Employee cannot be of type ${type}`
  }
}
```

子类中获取类型码的访问函数——get type函数——仍然留着。通常我会希望把这些函数也干掉，不过可能需要多花点儿时间，因为有其他函数使用了它们。我会用以多态取代条件表达式（272）和函数下移（359）来处理这些访问函数。到某个时候，已经没有代码使用类型码的访问函数了，我再用移除死代码（237）给它们送终。

范例：使用间接继承

还是前面这个例子，我们回到最起初的状态，不过这次我已经有了“全职员工”和“兼职员工”两个子类，所以不能再根据员工类别代码创建子类了。另外，我可能需要允许员工类别动态调整，这也会导致不能使用直接继承的方案。

class Employee...

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}
validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}
get type()    {return this._type;}
set type(arg) {this._type = arg;}

get capitalizedType() {

return this._type.charAt(0).toUpperCase() + this._type.substr
}
toString() {
  return `${this._name} (${this.capitalizedType})`;
}
```

这次的toString函数要更复杂一点，以便稍后展示用。

首先，我用以对象取代基本类型（174）包装类型码。

```
class EmployeeType {
  constructor(aString) {
    this._value = aString;
  }
  toString() {return this._value;}
}
```

class Employee...

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this.type = type;
}
validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}
get typeString()  {return this._type.toString();}
get type()    {return this._type;}
set type(arg) {this._type = new EmployeeType(arg);}

get capitalizedType() {
  return this.typeString.charAt(0).toUpperCase()
    + this.typeString.substr(1).toLowerCase();
}
toString() {
  return `${this._name} (${this.capitalizedType})`;
}
```

然后使用以子类取代类型码（362）的老套路，把员工类别代码变成子类。

class Employee...

```
set type(arg) {this._type = Employee.createEmployeeType(arg);

  static createEmployeeType(aString) {
    switch(aString) {
      case "engineer": return new Engineer();
      case "manager":  return new Manager ();
      case "salesman": return new Salesman();

default: throw new Error(`Employee cannot be of type ${aStrin
    }
  }

class EmployeeType {
}
class Engineer extends EmployeeType {
  toString() {return "engineer";}
}
class Manager extends EmployeeType {
  toString() {return "manager";}
}
class Salesman extends EmployeeType {
  toString() {return "salesman";}
}
```

如果重构到此为止的话，空的EmployeeType类可以去掉。但我更愿意留着它，用来明确表达各个子类之间的关系。并且有一个超类，也方便把其他行为搬移进去，例如我专门放在toString函数里的“名字大写”逻辑，就可以搬到超类。

class Employee...

```
toString() {
  return `${this._name} (${this.type.capitalizedName})`;
}
```

class EmployeeType...

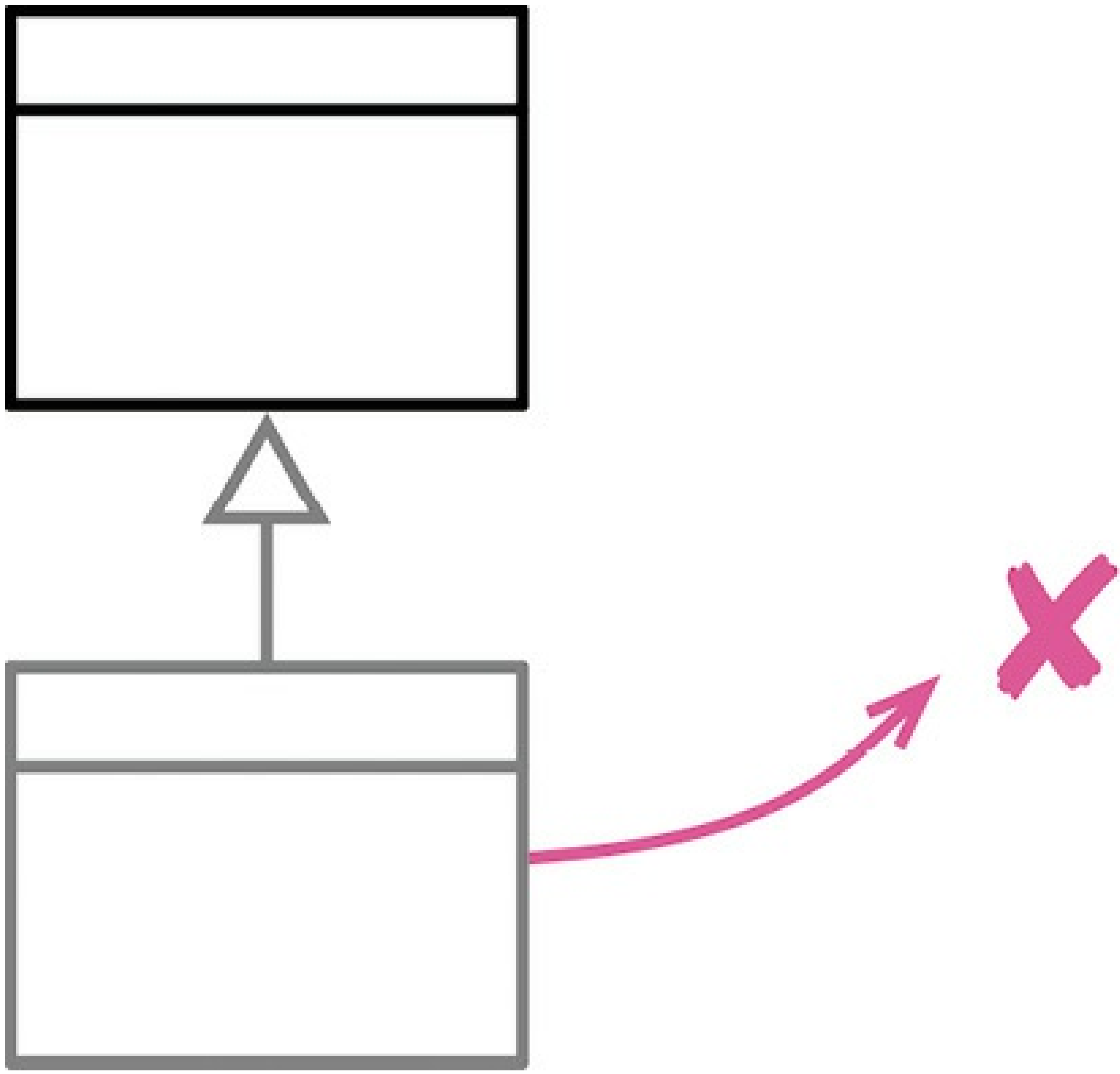
```
get capitalizedName() {
  return this.toString().charAt(0).toUpperCase()
    + this.toString().substr(1).toLowerCase();
}
```

熟悉本书第1版的读者大概能看出，这个例子来自第1版的以State/Strategy取代类型码重构手法。现在我认为这是以间接继承的方式使用以子类取代类型码，所以就不再将其作为一个单独的重构手法了。（而且我也一直不喜欢那个老重构手法的名字。）

12.7　移除子类（Remove Subclass）

曾用名：以字段取代子类（Replace Subclass with Fields）

反向重构：以子类取代类型码（362）



```
class Person {
  get genderCode() {return "X";}
}
class Male extends Person {
  get genderCode() {return "M";}
}
class Female extends Person {
  get genderCode() {return "F";}
}
```

```
class Person {
  get genderCode() {return this._genderCode;}
}
```

动机

子类很有用，它们为数据结构的多样和行为的多态提供支持，它们是针对差异编程的好工具。但随着软件的演化，子类所支持的变化可能会被搬移到别处，甚至完全去除，这时子类就失去了价值。有时添加子类是为了应对未来的功能，结果构想中的功能压根没被构造出来，或者用了另一种方式构造，使该子类不再被需要了。

子类存在着就有成本，阅读者要花心思去理解它的用意，所以如果子类的用处太少，就不值得存在了。此时，最好的选择就是移除子类，将其替换为超类中的一个字段。

做法

- 使用以工厂函数取代构造函数（334），把子类的构造函数包装到超类的工厂函数中。

如果构造函数的客户端用一个数组字段来决定实例化哪个子类，可以把这个判断逻辑放到超类的工厂函数中。

- 如果有任何代码检查子类的类型，先用提炼函数（106）把类型检查逻辑包装起来，然后用搬移函数（198）将其搬到超类。每次修改后执行测试。
- 新建一个字段，用于代表子类的类型。
- 将原本针对子类的类型做判断的函数改为使用新建的类型字段。
- 删除子类。
- 测试。

本重构手法常用于一次移除多个子类，此时需要先把这些子类都封装起来（添加工厂函数、搬移类型检查），然后再逐个将它们折叠到超类中。

范例

一开始，代码中遗留了两个子类。

class Person...

```
constructor(name) {
  this._name = name;
}
get name()    {return this._name;}
get genderCode() {return "X";}
// snip

class Male extends Person {
  get genderCode() {return "M";}
}

class Female extends Person {
  get genderCode() {return "F";}
}
```

如果子类就干这点儿事，那真的没必要存在。不过，在移除子类之前，通常有必要检查使用方代码是否有依赖于特定子类的行为，这样的行为需要被搬移到子类中。在这个例子里，我找到一些客户端代码基于子类的类型做判断，不过这也不足以成为保留子类的理由。

客户端...

```
const numberOfMales = people.filter(p => p instanceof Male).l
```

每当想要改变某个东西的表现形式时，我会先将当下的表现形式封装起来，从而尽量减小对客户端代码的影响。对于“创建子类对象”而言，封装的方式就是以工厂函数取代构造函数（334）。在这里，实现工厂有两种方式。

最直接的方式是为每个构造函数分别创建一个工厂函数。

```
function createPerson(name) {
  return new Person(name);
}
function createMale(name) {
  return new Male(name);
}
function createFemale(name) {
  return new Female(name);
}
```

虽然这是最直接的选择，但这样的对象经常是从输入源加载出来，直接根据性别代码创建对象。

```
function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    let p;
```

```
    switch (aRecord.gender) {
      case 'M': p = new Male(aRecord.name); break;
      case 'F': p = new Female(aRecord.name); break;
      default: p = new Person(aRecord.name);
    }
    result.push(p);
  });
  return result;
}
```

有鉴于此，我觉得更好的办法是先用提炼函数（106）把“选择哪个类来实例化”的逻辑提炼成工厂函数。

```
function createPerson(aRecord) {
  let p;
  switch (aRecord.gender) {
    case 'M': p = new Male(aRecord.name); break;
    case 'F': p = new Female(aRecord.name); break;
    default: p = new Person(aRecord.name);
  }
  return p;
}
function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    result.push(createPerson(aRecord));
  });
  return result;
}
```

提炼完工厂函数后，我会对这两个函数做些清理。先用内联变量（123）简化createPerson函数：

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Male  (aRecord.name);
    case 'F': return new Female(aRecord.name);
    default:  return new Person(aRecord.name);
  }
}
```

再用以管道取代循环（231）简化loadFromInput函数：

```
function loadFromInput(data) {
  return data.map(aRecord => createPerson(aRecord));
}
```

工厂函数封装了子类的创建逻辑，但代码中还有一处用到instanceof运算符——这从来不会是什么好味道。我用提炼函数（106）把这个类型检查逻辑提炼出来。

客户端...

```
const numberOfMales = people.filter(p => isMale(p)).length;

function isMale(aPerson) {return aPerson instanceof Male;}
```

然后用搬移函数（198）将其移到Person类。

class Person...

```
get isMale() {return this instanceof Male;}
```

客户端...

```
const numberOfMales = people.filter(p => p.isMale).length;
```

重构到这一步，所有与子类相关的知识都已经安全地包装在超类和工厂函数中。（对于“超类引用子类”这种情况，通常我会很警惕，不过这段代码用不了一杯茶的工夫就会被干掉，所以也不用太担心。）

现在，添加一个字段来表示子类之间的差异。既然有来自别处的一个类型代码，直接用它也无妨。

class Person...

```
constructor(name, genderCode) {
  this._name = name;
  this._genderCode = genderCode || "X";
}
```

```
get genderCode() {return this._genderCode;}
```

在初始化时先将其设置为默认值。（顺便说一句，虽然大多数人可以归类为男性或女性，但确实有些人不是这两种性别中的任何一种。忽视这些人的存在，是一个常见的建模错误。）

首先从“男性”的情况开始，将相关逻辑折叠到超类中。为此，首先要修改工厂函数，令其返回一个Person对象，然后修改所有instanceof检查逻辑，改为使用性别代码字段。

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Female(aRecord.name);
    default: return new Person(aRecord.name);
  }
}
```

class Person...

```
get isMale() {return "M" === this._genderCode;}
```

此时我可以测试，删除Male子类，再次测试，

然后对Female子类也如法炮制。

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default: return new Person(aRecord.name);
  }
}
```

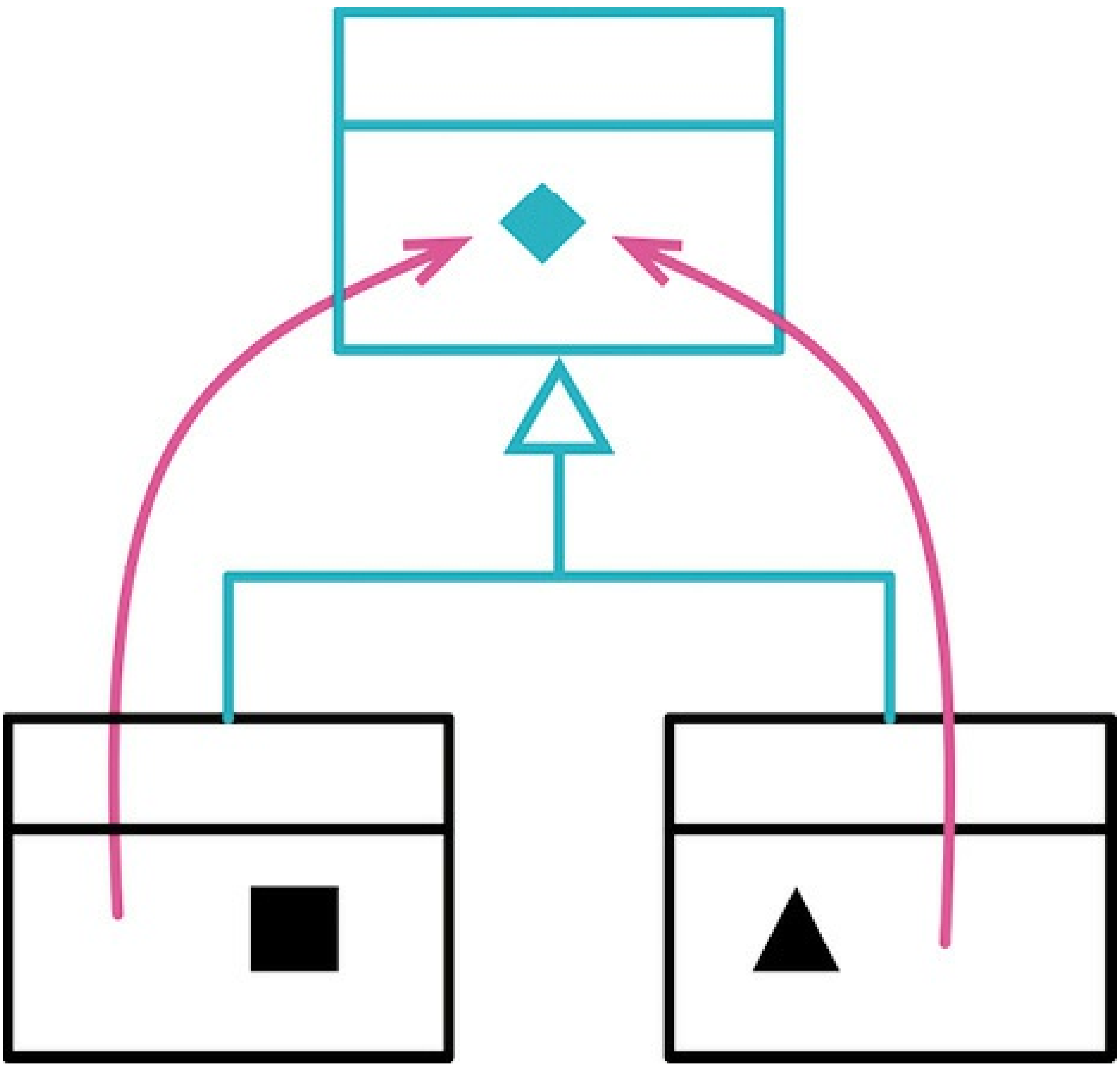
类型代码的分配有点儿失衡，默认情况没有类型代码，这种情况让我很烦心。未来阅读代码的人会一直好奇背后的原因。所以我更愿意现在做点儿修改，给所有情况都平等地分配类型代码——只要不会引入额外的复杂性就好。

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default: return new Person(aRecord.name, "X");
  }
}
```

class Person...

```
constructor(name, genderCode) {
  this._name = name;
  this._genderCode = genderCode || "X";
}
```

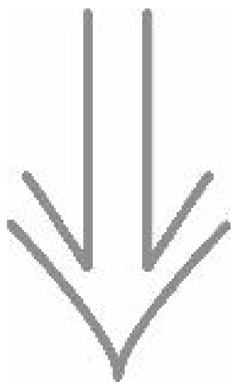
12.8 提炼超类（Extract Superclass）



动机

```
class Department {
  get totalAnnualCost() {...}
  get name() {...}
  get headCount() {...}
}

class Employee {
  get annualCost() {...}
  get name() {...}
  get id() {...}
}
```



```
class Party {
  get name() {...}
  get annualCost() {...}
}

class Department extends Party {
  get annualCost() {...}
  get headCount() {...}
}

class Employee extends Party {
  get annualCost() {...}
  get id() {...}
}
```

动机

如果我看见两个类在做相似的事，可以利用基本的继承机制把它们的相似之处提炼到超类。我可以用字段上移（353）把相同的数据搬到超类，用函数上移（350）搬移相同的行为。

很多技术作家在谈到面向对象时，认为继承必须预先仔细计划，应该根据“真实世界”的分类结构建立对象模型。真实世界的分类结构可以作为设计继承关系的提示，但还有很多时候，合理的继承关系是在程序演化的过程中才浮现出来的：我发现了一些共同元素，希望把它们抽取到一处，于是就有了继承关系。

另一种选择就是提炼类（182）。这两种方案之间的选择，其实就是继承和委托之间的选择，总之目的都是把重复的行为收拢一处。提炼超类通常是比较简单的做法，所以我会首选这个方案。即便选错了，也总有以委托取代超类（399）这瓶后悔药可吃。

做法

- 为原本的类新建一个空白的超类。

如果需要的话，用改变函数声明（124）调整构造函数的签名。

- 测试。
- 使用构造函数本体上移（355）、函数上移（350）和字段上移（353）手法，逐一将子类的共同元素上移到超类。
- 检查留在子类中的函数，看它们是否还有共同的成分。如果有，可以先用提炼函数（106）将其提炼出来，再用函数上移（350）搬到超类。
- 检查所有使用原本的类的客户端代码，考虑将其调整为使用超类的接口。

范例

下面这两个类，仔细考虑之下，是有一些共同之处的——它们都有名字（name），也都有月度成本（monthly cost）和年度成本（annual cost）的概念：

```
class Employee {
  constructor(name, id, monthlyCost) {
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
```

```
  get monthlyCost() {return this._monthlyCost;}
  get name() {return this._name;}
  get id() {return this._id;}

  get annualCost() {
    return this.monthlyCost * 12;
  }
}

class Department {
  constructor(name, staff){
    this._name = name;
    this._staff = staff;
  }
  get staff() {return this._staff.slice();}
  get name() {return this._name;}

  get totalMonthlyCost() {
    return this.staff
      .map(e => e.monthlyCost)
      .reduce((sum, cost) => sum + cost);
  }
  get headCount() {
    return this.staff.length;
  }
  get totalAnnualCost() {
    return this.totalMonthlyCost * 12;
  }
}
```

可以为它们提炼一个共同的超类，更明显地表达出它们之间的共同行为。

首先创建一个空的超类，让原来的两个类都继承这个新的类。

```
class Party {}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
```

```
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // rest of class...
class Department extends Party {
  constructor(name, staff){
    super();
    this._name = name;
    this._staff = staff;
  }
  // rest of class...
```

在提炼超类时，我喜欢先从数据开始搬移，在JavaScript中就需要修改构造函数。我先用字段上移（353）把name字段搬到超类中。

class Party...

```
constructor(name){
  this._name = name;
}
```

class Employee...

```
constructor(name, id, monthlyCost) {
  super(name);
  this._id = id;
  this._monthlyCost = monthlyCost;
}
```

class Department...

```
constructor(name, staff){
  super(name);
  this._staff = staff;
}
```

把数据搬到超类的同时，可以用函数上移（350）把相关的函数也一起搬移。首先是name函数：

class Party...

```
get name() {return this._name;}
```

class Employee...

```
get name() {return this._name;}
```

class Department...

```
get name() {return this._name;}
```

有两个函数实现非常相似。

class Employee...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

class Department...

```
get totalAnnualCost() {
  return this.totalMonthlyCost * 12;
}
```

它们各自使用的函数`monthlyCost`和`totalMonthlyCost`名字和实现都不同，但意图却是一致。我可以用改变函数声明（124）将它们的名字统一。

class Department...

```
get totalAnnualCost() {
  return this.monthlyCost * 12;
}

get monthlyCost() { ... }
```

然后对计算年度成本的函数也做相似的改名：

class Department...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

现在可以用函数上移（350）把这个函数搬到超类了。

class Party...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

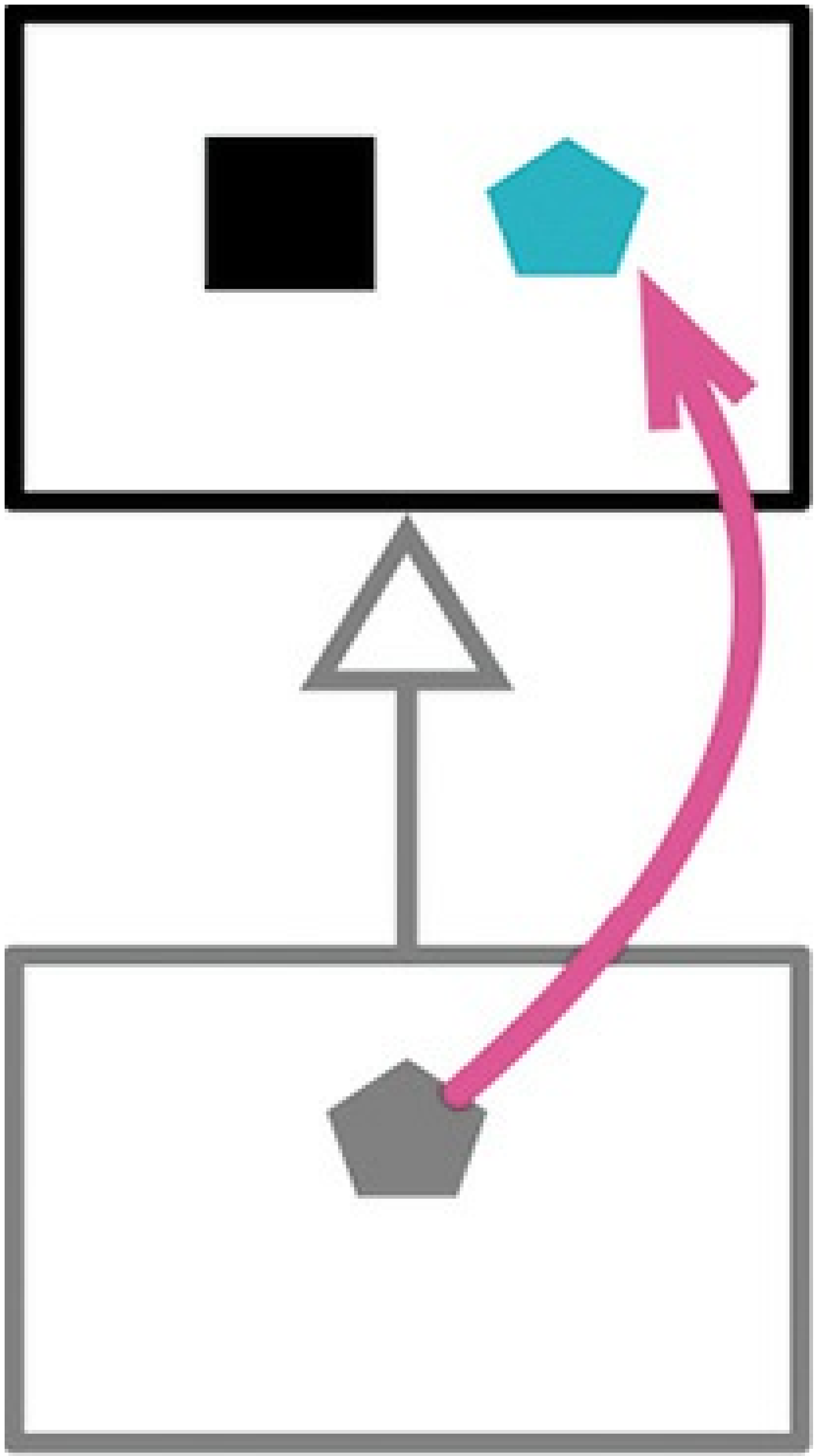
class Employee...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

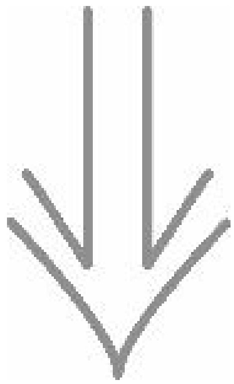
class Department...

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

12.9　折叠继承体系（Collapse Hierarchy）



```
class Employee {...}
class Salesman extends Employee {...}
```



```
class Employee {...}
```

动机

在重构类继承体系时，我经常把函数和字段上下移动。随着继承体系的演化，我有时会发现一个类与其超类已经没多大差别，不值得再作为独立的类存在。此时我就会把超类和子类合并起来。

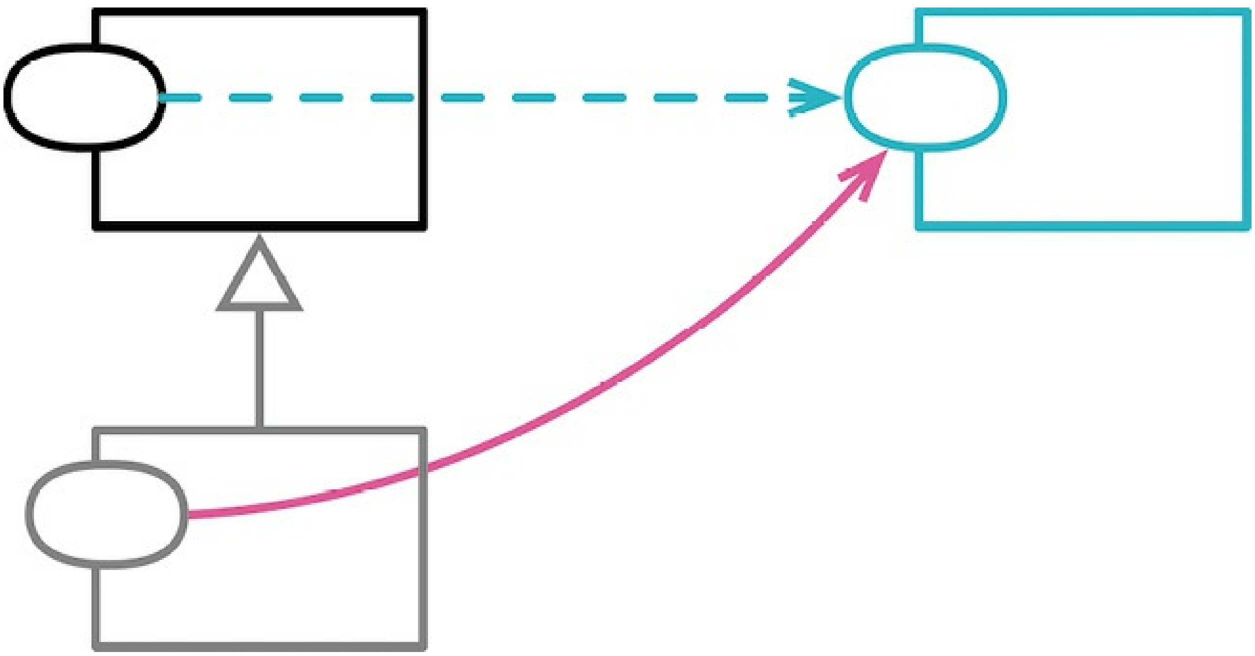
做法

- 选择想移除的类：是超类还是子类？

  我选择的依据是看哪个类的名字放在未来更有意义。如果两个名字都不够好，我就随便挑一个。

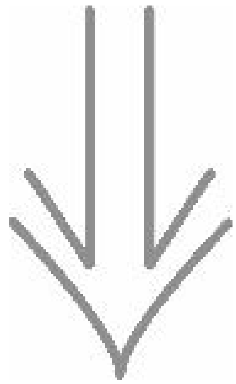
- 使用字段上移（353）、字段下移（361）、函数上移（350）和函数下移（359），把所有元素都移到同一个类中。
- 调整即将被移除的那个类的所有引用点，令它们改而引用合并后留下的类。
- 移除我们的目标；此时它应该已经成为一个空类。
- 测试。

12.10　以委托取代子类（Replace Subclass with Delegate）



```
class Order {
  get daysToShip() {
    return this._warehouse.daysToShip;
  }
}

class PriorityOrder extends Order {
  get daysToShip() {
    return this._priorityPlan.daysToShip;
  }
}
```



```
class Order {
  get daysToShip() {
    return (this._priorityDelegate)
      ? this._priorityDelegate.daysToShip
      : this._warehouse.daysToShip;
  }
}

class PriorityOrderDelegate {
  get daysToShip() {
    return this._priorityPlan.daysToShip
  }
}
```

动机

如果一个对象的行为有明显的类别之分，继承是很自然的表达方式。我可以把共用的数据和行为放在超类中，每个子类根据需要覆写部分特性。在面向对象语言中，继承很容易实现，因此也是程序员熟悉的机制。

但继承也有其短板。最明显的是，继承这张牌只能打一次。导致行为不同的原因可能有多种，但继承只能用于处理一个方向上的变化。比如说，我可能希望“人”的行为根据“年龄段”不同，并且根据“收入水平”不同。使用继承的话，子类可以是“年轻人”和“老人”，也可以是“富人”和“穷人”，但不能同时采用两种继承方式。

更大的问题在于，继承给类之间引入了非常紧密的关系。在超类上做任何修改，都很可能破坏子类，所以我必须非常小心，并且充分理解子类如何从超类派生。如果两个类的逻辑分处不同的模块、由不同的团队负责，问题就会更麻烦。

这两个问题用委托都能解决。对于不同的变化原因，我可以委托给不同的类。委托是对象之间常规的关系。与继承关系相比，使用委托关系时接口更清晰、耦合更少。因此，继承关系遇到问题时运用以委托取代子类是常见的情况。

有一条流行的原则：“对象组合优于类继承”（“组合”跟“委托”是同一回事）。很多人把这句话解读为“继承有害”，并因此声称绝不应该使用继承。我经常使用继承，部分是因为我知道，如果稍后需要改变，我总可以使用以委托取代子类。继承是一种很有价值的机制，大部分时候能达到效果，不会带来问题。所以我会从继承开始，如果开始出

现问题，再转而使用委托。这种用法与前面说的原则实际上是一致的——这条出自名著《设计模式》[gof]的原则解释了如何让继承和组合协同工作。这条原则之所以强调“组合优于继承”，其实是对彼时继承常被滥用的回应。

熟悉《设计模式》一书的读者可以这样来理解本重构手法，就是用状态（State）模式或者策略（Strategy）模式取代子类。这两个模式在结构上是相同的，都是由宿主对象把责任委托给另一个继承体系。以委托取代子类并非总会需要建立一个继承体系来接受委托（下面第一个例子就没有），不过建立一个状态或策略的继承体系经常都是有用的。

做法

- 如果构造函数有多个调用者，首先用以工厂函数取代构造函数（334）把构造函数包装起来。
- 创建一个空的委托类，这个类的构造函数应该接受所有子类特有的数据项，并且经常以参数的形式接受一个指回超类的引用。
- 在超类中添加一个字段，用于安放委托对象。
- 修改子类的创建逻辑，使其初始化上述委托字

段，放入一个委托对象的实例。

这一步可以在工厂函数中完成，也可以在构造函数中完成（如果构造函数有足够的信息以创建正确的委托对象的话）。

- 选择一个子类中的函数，将其移入委托类。
- 使用搬移函数（198）手法搬移上述函数，不要删除源类中的委托代码。

如果这个方法用到的其他元素也应该被移入委托对象，就把它们一并搬移。如果它用到的元素应该留在超类中，就在委托对象中添加一个字段，令其指向超类的实例。

- 如果被搬移的源函数还在子类之外被调用了，就把留在源类中的委托代码从子类移到超类，并在委托代码之前加上卫语句，检查委托对象存在。如果子类之外已经没有其他调用者，就用移除死代码（237）去掉已经没人使用的委托代码。

如果有多个委托类，并且其中的代码出现了重复，就使用提炼超类（375）手法消除重复。此时如果默认行为已经被移入了委托类的超类，源超类的委托函数就不再需要卫语句了。

- 测试。
- 重复上述过程，直到子类中所有函数都搬到委托类。
- 找到所有调用子类构造函数的地方，逐一将其改为使用超类的构造函数。
- 测试。
- 运用移除死代码（237）去掉子类。

范例

下面这个类用于处理演出（show）的预订（booking）。

class Booking...

```
constructor(show, date) {
  this._show = show;
```

```
    this._date = date;
  }
```

它有一个子类，专门用于预订高级（premium）票，这个子类要考虑各种附加服务（extra）。

class PremiumBooking extends Booking...

```
  constructor(show, date, extras) {
    super(show, date);
    this._extras = extras;
  }
```

PremiumBooking类在超类基础上做了好些改变。在这种“针对差异编程”（programming-by-difference）的风格中，子类常会覆写超类的方法，有时还会添加只对子类有意义的新方法。我不打算讨论所有差异点，只选几处有意思的案例来分析。

先来看一处简单的覆写。常规票在演出结束后会有“对话创作者”环节（talkback），但只在非高峰日提供这项服务。

class Booking...

```
get hasTalkback() {

return this._show.hasOwnProperty('talkback') && !this.isPeakD
}
```

PremiumBooking覆写了这个逻辑，任何一天都提供与创作者的对话。

class PremiumBooking...

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

定价逻辑也是相似的覆写，不过略有不同：PremiumBooking调用了超类中的方法。

class Booking...

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

class PremiumBooking...

```
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee
}
```

最后一个例子是PremiumBooking提供了一个超类中没有的行为。

class PremiumBooking...

```
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this.isPea
}
```

继承在这个例子中工作良好。即使不了解子类，我同样也可以理解超类的逻辑。子类只描述自己与超类的差异——既避免了重复，又清晰地表述了自己引入的差异。

说真的，它也并非如此完美。超类的一些结构只在特定的子类存在时才有意义——有些函数的组织方式完全就是为了方便覆写特定类型的行为。所以，尽管大部分时候我可以修改超类而不必理解子类，但如果刻意不关注子类的存在，在修改超类时偶尔有可能会破坏子类。不过，如果这种“偶尔”发生得不太频繁，继承就还是划算的——只要我有良好的测试，当子类被破坏时就能及时发现。

那么，既然情况还算不坏，为什么我想用以委托取代子类来做出改变呢？因为继承只能使用一次，如果我有别的原因想使用继承，并且这个新的原因比“高级预订”更有必要，就需要换一种方式来处理高级预订。另外，我可能需要动态地把普通预订升级成高级预订，例如提供`aBooking.bePremium()`这样一个函数。有时我可以新建一个对象（就好像通过HTTP请求从服务器端加载全新的数据），从而避免“对象本身升级”的问题。但有时我需要修改数据本身的结构，而不重建整个数据结构。如果一个`Booking`对象被很多地方引用，也很难将其整个替换掉。此时，就有必要允许在“普通预订”和“高级预订”之间来回转换。

当这样的需求积累到一定程度时，我就该使用以委托取代子类了。现在客户端直接调用两个类的构造函数来创建不同的预订。

进行普通预订的客户端

```
aBooking = new Booking(show,date);
```

进行高级预订的客户端

```
aBooking = new PremiumBooking(show, date, extras);
```

去除子类会改变对象创建的方式，所以我要先用以工厂函数取代构造函数（334）把构造函数封装起来。

顶层作用域...

```
function createBooking(show, date) {
  return new Booking(show, date);
}
function createPremiumBooking(show, date, extras) {
  return new PremiumBooking (show, date, extras);
}
```

进行普通预订的客户端

```
aBooking = createBooking(show, date);
```

进行高级预订的客户端

```
aBooking = createPremiumBooking(show, date, extras);
```

　　然后新建一个委托类。这个类的构造函数参数有两部分：首先是指向Booking对象的反向引用，随后是只有子类才需要的那些数据。我需要传入反向引用，是因为子类的几个函数需要访问超类中的数据。有继承关系的时候，访问这些数据很容易；而在委托关系中，就得通过反向引用来访问。

class PremiumBookingDelegate...

```
constructor(hostBooking, extras) {
  this._host = hostBooking;
  this._extras = extras;
}
```

　　现在可以把新建的委托对象与Booking对象关联起来。在“创建高级预订”的工厂函数中修改即可。

顶层作用域...

```
function createPremiumBooking(show, date, extras) {
  const result = new PremiumBooking (show, date, extras);
  result._bePremium(extras);
  return result;
}
```

class Booking...

```
_bePremium(extras) {
  this._premiumDelegate = new PremiumBookingDelegate(this, ex
}
```

_bePremium函数以下划线开头，表示这个函数不应该被当作Booking类的公共接口。当然，如果最终我们希望允许普通预订转换成高级预订，这个函数也可以成为公共接口。

或者我也可以在Booking类的构造函数中构建它与委托对象之间的联系。为此，我需要以某种方式告诉构造函数“这是一个高级预订”：可以通过一个额外的参数，也可以直接通过extras参数来表示（如果我能确定这个参数只有高级预订才会用到的话）。不过我还是更愿意在工厂函数中构建这层联系，因为这样可以把意图表达得更明确。

结构设置好了，现在该动手搬移行为了。我首先考虑hasTalkback函数简单的覆写逻辑。现在的代码如下。

class Booking...

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback') && !this.isPea
}
```

class PremiumBooking...

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

我用搬移函数（198）把子类中的函数搬到委托类中。为了让它适应新家，原本访问超类中数据的代码，现在要改为调用`_host`对象。

class PremiumBookingDelegate...

```
get hasTalkback() {
  return this._host._show.hasOwnProperty('talkback');
}
```

class PremiumBooking...

```
get hasTalkback() {
  return this._premiumDelegate.hasTalkback;
}
```

测试，确保一切正常，然后把子类中的函数删掉：

class PremiumBooking...

```
get hasTalkback() {
  return this._premiumDelegate.hasTalkback;
}
```

再次测试，现在应该有一些测试失败，因为原本有些代码会用到子类上的`hasTalkback`函数。

现在我要修复这些失败的测试：在超类的函数中添加适当的分发逻辑，如果有代理对象存在就使用代理对象。这样，这一步重构就算完成了。

class Booking...

```
get hasTalkback() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.hasTalkback
    : this._show.hasOwnProperty('talkback') && !this.isPeakDa
}
```

下一个要处理的是basePrice函数。

class Booking...

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

class PremiumBooking...

```
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee
}
```

情况大致相同，但有一点儿小麻烦：子类中调用了超类中的同名函数（在这种“子类扩展超类行为”的用法中，这种情况很常见）。把子类的代码移到委托类时，需要继续调用超类的逻辑——但我不能直接调用this._host.basePrice，这会导致无穷递归，因为_host对象就是PremiumBooking对象自己。

有两个办法来处理这个问题。一种办法是，可以用提炼函数（106）把“基本价格”的计算逻辑提

炼出来，从而把分发逻辑与价格计算逻辑拆开。（剩下的操作就跟前面的例子一样了。）

class Booking...

```
get basePrice() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.basePrice
    : this._privateBasePrice;
}

get _privateBasePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

class PremiumBookingDelegate...

```
get basePrice() {
   return Math.round(this._host._privateBasePrice + this._extr
}
```

另一种办法是，可以重新定义委托对象中的函数，使其成为基础函数的扩展。

class Booking...

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return (this._premiumDelegate)
    ? this._premiumDelegate.extendBasePrice(result)
    : result;
}
```

class PremiumBookingDelegate...

```
extendBasePrice(base) {
  return Math.round(base + this._extras.premiumFee);
}
```

两种办法都可行，我更偏爱后者一点儿，因为需要的代码较少。

最后一个例子是一个只存在于子类中的函数。

class PremiumBooking...

```
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this.isPea
}
```

我把它从子类移到委托类。

class PremiumBookingDelegate...

```
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this._host
}
```

然后在Booking类中添加分发逻辑。

class Booking...

```
get hasDinner() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.hasDinner
    : undefined;
}
```

在JavaScript中，如果尝试访问一个没有定义的属性，就会得到undefined，所以我在这个函数中也这样做。（尽管我直觉认为应该抛出错误，我所熟悉的其他面向对象动态语言就是这样做的。）

所有的行为都从子类中搬移出去之后，我就可以修改工厂函数，令其返回超类的实例。再次运行测试，确保一切都运转良好，然后我就可以删除子类。

顶层作用域…

```
function createPremiumBooking(show, date, extras) {
  const result = new PremiumBooking (show, date, extras);
  result._bePremium(extras);
  return result;
}

~~class PremiumBooking extends Booking ...~~
```

只看这个重构本身，我并不觉得代码质量得到了提升。继承原本很好地应对了需求场景，换成委托以后，我增加了分发逻辑、双向引用，复杂度上升不少。不过这个重构可能还是值得的，因为现在“是否高级预订”这个状态可以改变了，并且我也可以用继承来达成其他目的了。如果有这些需求的话，去除原有的继承关系带来的损失可能还是划算的。

范例：取代继承体系

前面的例子展示了如何用以委托取代子类去除单个子类。还可以用这个重构手法去除整个继承体系。

```
function createBird(data) {
```

```
  switch (data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(data);
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}

class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
  }
  get name()  {return this._name;}

  get plumage() {
    return this._plumage || "average";
  }
  get airSpeedVelocity() {return null;}
}

class EuropeanSwallow extends Bird {
  get airSpeedVelocity() {return 35;}
}

class AfricanSwallow extends Bird {
  constructor(data) {
    super (data);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}

class NorwegianBlueParrot extends Bird {
  constructor(data) {
    super (data);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }

  get plumage() {
    if (this._voltage > 100) return "scorched";
```

```
    else return this._plumage || "beautiful";
  }
  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
}
```

上面这个关于鸟儿（bird）的系统很快要有一个大变化：有些鸟是“野生的”（wild），有些鸟是“家养的”（captive），两者之间的行为会有很大差异。这种差异可以建模为Bird类的两个子类：WildBird和CaptiveBird。但继承只能用一次，所以如果想用子类来表现“野生”和“家养”的差异，就得先去掉关于“不同品种”的继承关系。

在涉及多个子类时，我会一次处理一个子类，先从简单的开始——在这里，最简单的当属EuropeanSwallow（欧洲燕）。我先给它建一个空的委托类。

```
class EuropeanSwallowDelegate {
}
```

委托类中暂时还没有传入任何数据或反向引用。在这个例子里，我会在需要时再引入这些参数。

现在需要决定如何初始化委托字段。由于构造

函数接受的唯一参数data包含了所有的信息，我决定在构造函数中初始化委托字段。考虑到有多个委托对象要添加，我会建一个函数，其中根据类型码（data.type）来选择适当的委托对象。

class Bird...

```
constructor(data) {
  this._name = data.name;
  this._plumage = data.plumage;
  this._speciesDelegate = this.selectSpeciesDelegate(data);
}

  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate();
      default: return null;
    }
  }
```

结构设置完毕，我可以用搬移函数（198）把EuropeanSwallow的airSpeedVelocity函数搬到委托对象中。

class EuropeanSwallowDelegate...

```
get airSpeedVelocity() {return 35;}
```

class EuropeanSwallow...

```
get airSpeedVelocity() {return this._speciesDelegate.airSpeed
```

修改超类的airSpeedVelocity函数，如果发现有委托对象存在，就调用之。

class Bird...

```
get airSpeedVelocity() {
  return this._speciesDelegate ? this._speciesDelegate.airSpe
}
```

然后，删除子类。

```
class EuropeanSwallow extends Bird {
  get airSpeedVelocity() {return this._speciesDelegate.airSpe
}
```

顶层作用域...

```
function createBird(data) {
  switch (data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(data);
```

```
      case 'AfricanSwallow':
        return new AfricanSwallow(data);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrot(data);
      default:
        return new Bird(data);
    }
  }
```

接下来处理AfricanSwallow（非洲燕）子类。为它创建一个委托类，这次委托类的构造函数需要传入data参数。

class AfricanSwallowDelegate...

```
  constructor(data) {
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
```

class Bird…

```
  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate();
      case 'AfricanSwallow':
        return new AfricanSwallowDelegate(data);
      default: return null;
    }
  }
```

同样用搬移函数（198）把airSpeedVelocity搬到委托类中。

class AfricanSwallowDelegate...

```
get airSpeedVelocity() {
  return 40 - 2 * this._numberOfCoconuts;
}
```

class AfricanSwallow...

```
get airSpeedVelocity() {
  return this._speciesDelegate.airSpeedVelocity;
}
```

再删掉AfricanSwallow子类。

```
class AfricanSwallow extends Bird {
  // all of the body ...
  }

function createBird(data) {
  switch (data.type) {
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
```

```
}
```

接下来是NorwegianBlueParrot（挪威蓝鹦鹉）子类。创建委托类和搬移airSpeed Velocity函数的步骤都跟前面一样，所以我直接展示结果好了。

class Bird...

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data);
    default: return null;
  }
}
```

class NorwegianBlueParrotDelegate...

```
constructor(data) {
  this._voltage = data.voltage;
  this._isNailed = data.isNailed;
}
get airSpeedVelocity() {
  return (this._isNailed) ? 0 : 10 + this._voltage / 10;
}
```

一切正常。但NorwegianBlueParrot还覆写了plumage属性，前面两个例子则没有。首先我还是用搬移函数（198）把plumage函数搬到委托类中，这一步不难，不过需要修改构造函数，放入对Bird对象的反向引用。

class NorwegianBlueParrot...

```
get plumage() {
  return this._speciesDelegate.plumage;
}
```

class NorwegianBlueParrotDelegate...

```
get plumage() {
  if (this._voltage > 100) return "scorched";
  else return this._bird._plumage || "beautiful";
}

constructor(data, bird) {
  this._bird = bird;
  this._voltage = data.voltage;
  this._isNailed = data.isNailed;
}
```

class Bird...

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data, this);
    default: return null;
  }
}
```

麻烦之处在于如何去掉子类中的plumage函数。如果我像下面这么干就会得到一大堆错误，因为其他品种的委托类没有plumage这个属性。

class Bird...

```
get plumage() {
  if (this._speciesDelegate)
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

我可以做一个更明确的条件分发：

class Bird...

```
get plumage() {
  if (this._speciesDelegate instanceof NorwegianBlueParrotDel
```

```
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

不过我超级反感这种做法，希望你也能闻出同样的坏味道。像这样的显式类型检查几乎总是坏主意。

另一个办法是在其他委托类中实现默认的行为。

class Bird...

```
get plumage() {
  if (this._speciesDelegate)
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

class EuropeanSwallowDelegate...

```
get plumage() {
  return this._bird._plumage || "average";
}
```

class AfricanSwallowDelegate...

```
get plumage() {
  return this._bird._plumage || "average";
}
```

但这又造成了`plumage`默认行为的重复。如果这还不够糟糕的话，还有一个“额外奖励”：构造函数中给`_bird`反向引用赋值的代码也会重复。

解决重复的办法，很自然，就是继承——用提炼超类（375）从各个代理类中提炼出一个共同继承的超类。

```
class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
  get plumage() {
    return this._bird._plumage || "average";
  }

class EuropeanSwallowDelegate extends SpeciesDelegate {

class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
}

class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
}
```

有了共同的超类以后，就可以把SpeciesDelegate字段默认设置为这个超类的实例，并把Bird类中的默认行为搬移到SpeciesDelegate超类中。

class Bird...

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate(data, this);
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data, this);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data, this);
    default: return new SpeciesDelegate(data, this);
  }
}
// rest of bird’s code...

get plumage() {return this._speciesDelegate.plumage;}

get airSpeedVelocity() {return this._speciesDelegate.airSpeed
```

class SpeciesDelegate...

```
get airSpeedVelocity() {return null;}
```

我喜欢这种办法，因为它简化了Bird类中的委

托函数。我可以一目了然地看到哪些行为已经被委托给SpeciesDelegate，哪些行为还留在Bird类中。

这几个类最终的状态如下：

```
function createBird(data) {
  return new Bird(data);
}
class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
    this._speciesDelegate = this.selectSpeciesDelegate(data);
  }
  get name()    {return this._name;}
  get plumage() {return this._speciesDelegate.plumage;}

get airSpeedVelocity() {return this._speciesDelegate.airSpeed

  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate(data, this);
      case 'AfricanSwallow':
        return new AfricanSwallowDelegate(data, this);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrotDelegate(data, this);
      default: return new SpeciesDelegate(data, this);
    }
  }
  // rest of bird’s code...
}

class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
  get plumage() {
    return this._bird._plumage || "average";
  }
  get airSpeedVelocity() {return null;}
}
```

```
class EuropeanSwallowDelegate extends SpeciesDelegate {
  get airSpeedVelocity() {return 35;}
}

class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}

class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._bird._plumage || "beautiful";
  }
}
```

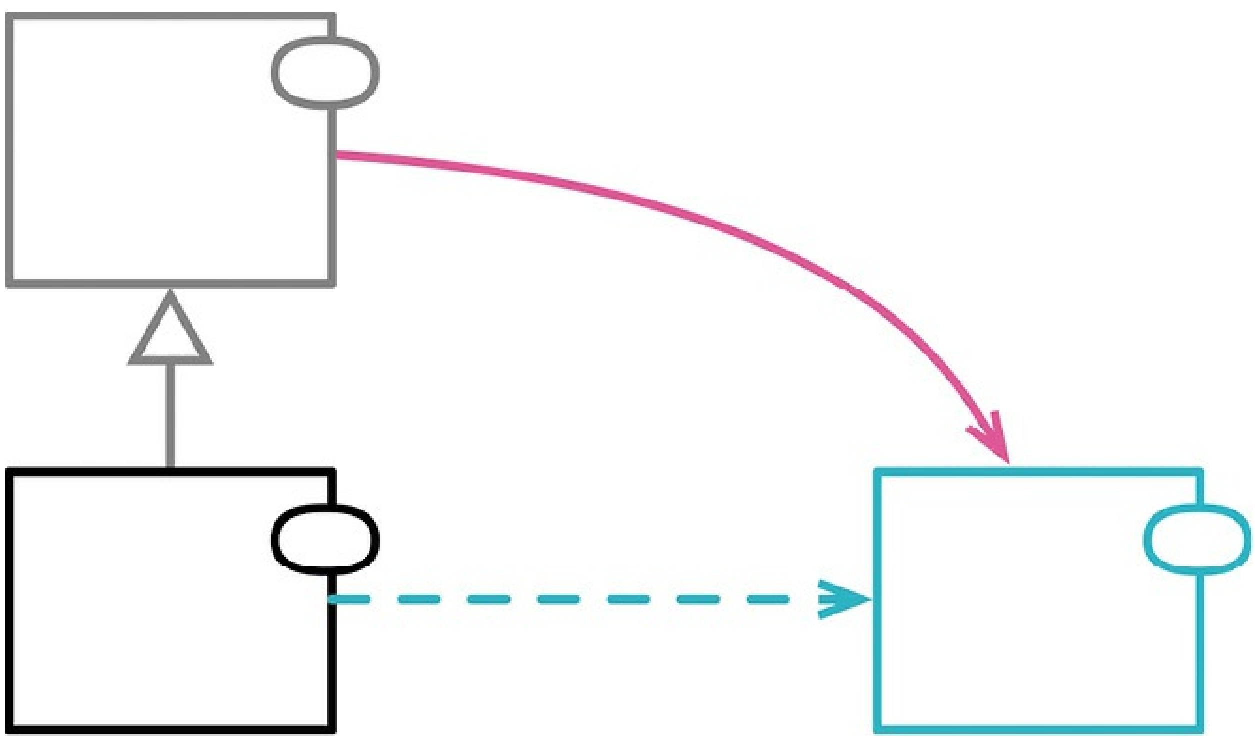
在这个例子中，我用一系列委托类取代了原来的多个子类，与原来非常相似的继承结构被转移到了SpeciesDelegate下面。除了给Bird类重新被继承的机会，从这个重构中我还有什么收获？新的继承体系范围更收拢了，只涉及各个品种不同的数据和行为，各个品种相同的代码则全都留在了Bird中，它未来的子类也将得益于这些共用的行为。

在前面的“演出预订”的例子中，我也可以采用同样的手法，创建一个委托超类。这样在Booking类中就不需要分发逻辑，直接调用委托对象即可，让继承关系来搞定分发。不过写到这儿，我要去吃晚饭了，就把这个练习留给读者吧。

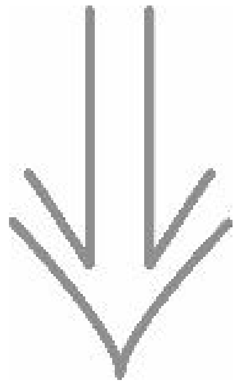
从这两个例子看来，“对象组合优于类继承”这句话更确切的表述可能应该是“审慎地组合使用对象组合与类继承，优于单独使用其中任何一种”——不过这就不太上口了。

12.11　以委托取代超类（Replace Superclass with Delegate）

曾用名：以委托取代继承（Replace Inheritance with Delegation）



```
class List {...}
class Stack extends List {...}
```



```
class Stack {
  constructor() {
    this._storage = new List();
  }
}
class List {...}
```

动机

在面向对象程序中，通过继承来复用现有功能，是一种既强大又便捷的手段。我只要继承一个已有的类，覆写一些功能，再添加一些功能，就能达成目的。但继承也有可能造成困扰和混乱。

在对象技术发展早期，有一个经典的误用继承的例子：让栈（stack）继承列表（list）。这个想法的出发点是想复用列表类的数据存储和操作能力。虽说复用是一件好事，但这个继承关系有问题：列表类的所有操作都会出现在栈类的接口上，然而其

中大部分操作对一个栈来说并不适用。更好的做法应该是把列表作为栈的字段，把必要的操作委派给列表就行了。

这就是一个用得上以委托取代超类手法的例子——如果超类的一些函数对子类并不适用，就说明我不应该通过继承来获得超类的功能。

除了“子类用得上超类的所有函数”之外，合理的继承关系还有一个重要特征：子类的所有实例都应该是超类的实例，通过超类的接口来使用子类的实例应该完全不出问题。假如我有一个车模（car model）类，其中有名称、引擎大小等属性，我可能想复用这些特性来表示真正的汽车（car），并在子类上添加VIN编号、制造日期等属性。然而汽车终归不是模型。这是一种常见而又经常不易察觉的建模错误，我称之为“类型与实例名不符实”（type-instance homonym）[mf-tih]。

在这两个例子中，有问题的继承招致了混乱和错误——如果把继承关系改为将部分职能委托给另一个对象，这些混乱和错误本是可以轻松避免的。使用委托关系能更清晰地表达“这是另一个东西，我只是需要用到其中携带的一些功能”这层意思。

即便在子类继承是合理的建模方式的情况下，如果子类与超类之间的耦合过强，超类的变化很容

易破坏子类的功能，我还是会使用以委托取代超类。这样做的缺点就是，对于宿主类（也就是原来的子类）和委托类（也就是原来的超类）中原本一样的函数，现在我必须在宿主类中挨个编写转发函数。不过还好，这种转发函数虽然写起来乏味，但它们都非常简单，几乎不可能出错。

有些人在这个方向上走得更远，他们建议完全避免使用继承，但我不同意这种观点。如果符合继承关系的语义条件（超类的所有方法都适用于子类，子类的所有实例都是超类的实例），那么继承是一种简洁又高效的复用机制。如果情况发生变化，继承不再是最好的选择，我也可以比较容易地运用以委托取代超类。所以我的建议是，首先（尽量）使用继承，如果发现继承有问题，再使用以委托取代超类。

做法

- 在子类中新建一个字段，使其引用超类的一个对象，并将这个委托引用初始化为超类的新实例。
- 针对超类的每个函数，在子类中创建一个转发函数，将调用请求转发给委托引用。每转发一

块完整逻辑，都要执行测试。

大多数时候，每转发一个函数就可以测试，但一对设值/取值必须同时转移，然后才能测试。

- 当所有超类函数都被转发函数覆写后，就可以去掉继承关系。

范例

我最近给一个古城里存放上古卷轴（scroll）的图书馆做了咨询。他们给卷轴的信息编制了一份目录（catalog），每份卷轴都有一个ID号，并记录了卷轴的标题（title）和一系列标签（tag）。

class CatalogItem...

```
constructor(id, title, tags) {
  this._id = id;
  this._title = title;
  this._tags = tags;
}
```

```
get id() {return this._id;}
get title() {return this._title;}
hasTag(arg) {return this._tags.includes(arg);}
```

这些古老的卷轴需要日常清扫，因此代表卷轴的`Scroll`类继承了代表目录项的`CatalogItem`类，并扩展出与“需要清扫”相关的数据。

class Scroll extends CatalogItem...

```
constructor(id, title, tags, dateLastCleaned) {
  super(id, title, tags);
  this._lastCleaned = dateLastCleaned;
}

needsCleaning(targetDate) {
  const threshold = this.hasTag("revered") ? 700 : 1500;
  return this.daysSinceLastCleaning(targetDate) > threshold ;
}
daysSinceLastCleaning(targetDate) {

return this._lastCleaned.until(targetDate, ChronoUnit.DAYS);
}
```

这就是一个常见的建模错误。真实存在的卷轴和只存在于纸面上的目录项，是完全不同的两种东西。比如说，关于“如何治疗灰鳞病”的卷轴可能有好几卷，但在目录上却只记录一个条目。

这样的建模错误很多时候可以置之不理。

像“标题”和“标签”这样的数据，我可以认为就是目录中数据的副本。如果这些数据从不发生改变，我完全可以接受这样的表现形式。但如果需要更新其中某处数据，我就必须非常小心，确保同一个目录项对应的所有数据副本都被正确地更新。

就算没有数据更新的问题，我还是希望改变这两个类之间的关系。把“目录项”作为“卷轴”的超类很可能会把未来的程序员搞迷糊，因此这是一个糟糕的模型。

我首先在`Scroll`类中创建一个属性，令其指向一个新建的`CatalogItem`实例。

class Scroll extends CatalogItem...

```
  constructor(id, title, tags, dateLastCleaned) {
    super(id, title, tags);
    this._catalogItem = new CatalogItem(id, title, tags);
    this._lastCleaned = dateLastCleaned;
  }
```

然后对于子类中用到所有属于超类的函数，我要逐一为它们创建转发函数。

class Scroll...

```
  get id() {return this._catalogItem.id;}
  get title() {return this._catalogItem.title;}
  hasTag(aString) {return this._catalogItem.hasTag(aString);}
```

最后去除Scroll与CatalogItem之间的继承关系。

```
class Scroll extends CatalogItem{
  constructor(id, title, tags, dateLastCleaned) {
    super(id, title, tags);
    this._catalogItem = new CatalogItem(id, title, tags);
    this._lastCleaned = dateLastCleaned;
  }
```

基本的以委托取代超类重构到这里就完成了，不过在这个例子中，我还有一点收尾工作要做。

前面的重构把CatalogItem变成了Scroll的一个组件：每个Scroll对象包含一个独一无二的CatalogItem对象。在使用本重构的很多情况下，这样处理就够了。但在这个例子中，更好的建模方式应该是：关于灰鳞病的一个目录项，对应于图书馆中的6份卷轴，因为这6份卷轴都是同一个标题。这实际上是要运用将值对象改为引用对象（256）。

不过在使用将值对象改为引用对象（256）之前，还有一个问题需要先修好。在原来的继承结构中，Scroll类使用了CatalogItem类的id字段来保

存自己的ID。但如果我把CatalogItem当作引用来处理，那么透过这个引用获得的ID就应该是目录项的ID，而不是卷轴的ID。也就是说，我需要在Scroll类上添加id字段，在创建Scroll对象时使用这个字段，而不是使用来自CatalogItem类的id字段。这一步既可以说是搬移，也可以说是拆分。

class Scroll...

```
constructor(id, title, tags, dateLastCleaned) {
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}

get id() {return this._id;}
```

用null作为ID值创建目录项，这种操作一般而言应该触发警报了，不过这只是我在重构过程中的临时状态，可以暂时忍受。等我重构完成，多个卷轴会指向一个共享的目录项，而后者也会有合适的ID。

当前Scroll对象是从一个加载程序中加载的。

加载程序...

```
const scrolls = aDocument
     .map(record => new Scroll(record.id,
                               record.catalogData.title,
                               record.catalogData.tags,

LocalDate.parse(record.lastCleaned)));
```

将值对象改为引用对象（256）的第一步是要找到或者创建一个仓库对象（repository）。我发现有一个仓库对象可以很容易地导入加载程序中，这个仓库对象负责提供`CatalogItem`对象，并用ID作为索引。我的下一项任务就是要想办法把这个ID值放进`Scroll`对象的构造函数。还好，输入数据中有这个值，不过之前一直被无视了，因为在使用继承的时候用不着。把这些信息都理清楚，我就可以运用改变函数声明（124），把整个目录对象以及目录项的ID都作为参数传给`Scroll`的构造函数。

加载程序...

```
const scrolls = aDocument
     .map(record => new Scroll(record.id,
                               record.catalogData.title,
                               record.catalogData.tags,

LocalDate.parse(record.lastCleaned),
                               record.catalogData.id,
                               catalog));
```

class Scroll...

```
constructor(id, title, tags, dateLastCleaned, catalogID, cata
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}
```

然后修改Scroll的构造函数，用传入的catalogID来查找对应的CatalogItem对象，并引用这个对象（而不再新建CatalogItem对象）。

class Scroll...

```
constructor(id, title, tags, dateLastCleaned, catalogID, cata
  this._id = id;
  this._catalogItem = catalog.get(catalogID);
  this._lastCleaned = dateLastCleaned;
}
```

Scroll的构造函数已经不再需要传入title和tags这两个参数了，所以我用改变函数声明（124）把它们去掉。

加载程序...

```
const scrolls = aDocument
     .map(record => new Scroll(record.id,
                              record.catalogData.title,
                              record.catalogData.tags,

LocalDate.parse(record.lastCleaned),
                              record.catalogData.id,
                              catalog));
```

class Scroll...

```
constructor(id, title, tags, dateLastCleaned, catalogID, cata
  this._id = id;
  this._catalogItem = catalog.get(catalogID);
  this._lastCleaned = dateLastCleaned;
}
```

参考文献

在以下网址可以找到这份参考文献的在线版本：https://martinfowler.com/books/refactoring-bibliography.html。本参考文献中很多条目都指向“bliki”——这是martinfowler.com网站的一个栏目，我在其中简要介绍了一些软件开发中用到的词汇。在撰写本书的过程中，我决定不把这些名词解释放在本书正文中，而是引导读者去bliki查看。

[Ambler & Sadalage] Scott W. Ambler and Pramod J. Sadalage. *Refactoring Databases*. Addison-Wesley, 2006. ISBN 0321293533. **1**

[Bazuzi] Jay Bazuzi. “Safely Extract a Method in Any C++ Code.”

[Beck SBPP] Kent Beck. *Smalltalk Best Practice Patterns*. Addison-Wesley, 1997. ISBN 013476904X.

[Feathers] Michael Feathers. *Working Effectively with Legacy Code*. Prentice-Hall, 2004. ISBN 0131177052. **2**

[Fields et al.] Jay Fields, Shane Harvie, and Martin Fowler. *Refactoring Ruby Edition*. Addison-Wesley, 2009. ISBN 0321603508.**3**

[Ford et al.] Neal Ford, Rebecca Parsons, and Patrick Kua. *Building Evolutionary Architectures*. O'Reilly, 2017. ISBN 1491986360.

[Forsgren et al.] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 2018. ISBN 1942788339.

[gof] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.**4**

[Harold] Elliotte Rusty Harold. *Refactoring HTML*. Addison-Wesley, 2008. ISBN 0321503635.**5**

[Kerievsky] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004. ISBN 0321213351. **6**

[mf-2h] Martin Fowler. "Bliki: TwoHardThings." https://martinfowler.com/bliki/TwoHardThings.html.

[mf-bba] Martin Fowler. “Bliki: BranchByAbstraction.” https://martinfowler.com/bliki/BranchByAbstraction.htm

[mf-cp] Martin Fowler. “Collection Pipeline.” https://martinfowler.com/articles/collection-pipeline/.

[mf-cqs] Martin Fowler. “Bliki: CommandQuerySeparation.” https://martinfowler.com/bliki/CommandQuerySeparati

[mf-cw] Martin Fowler. “Bliki: ClockWrapper.” https://martinfowler.com/bliki/ClockWrapper.html.

[mf-dsh] Martin Fowler. “Bliki: DesignStaminaHypothesis.” https://martinfowler.com/bliki/DesignStaminaHypothes

[mf-evodb] Pramod Sadalage and Martin Fowler. “Evolutionary Database Design.” https://martinfowler.com/articles/evodb.html.

[mf-fao] Martin Fowler. “Bliki: FunctionAsObject.” https://martinfowler.com/bliki/FunctionAsObject.html.

[mf-ft] Martin Fowler. “Form Template Method.” https://refactoring.com/catalog/formTemplateMethod.ht

[mf-lh] Martin Fowler. “Bliki: ListAndHash.” https://martinfowler.com/bliki/ListAndHash.html.

[mf-nm] Martin Fowler. “The New Methodology.” https://martinfowler.com/articles/newMethodology.htm

[mf-ogs] Martin Fowler. “Bliki: OverloadedGetterSetter.” https://martinfowler.com/bliki/OverloadedGetterSetter.h

[mf-pc] Danilo Sato. “Bliki: ParallelChange.” https://martinfowler.com/bliki/ParallelChange.html.

[mf-range] Martin Fowler. “Range.” https://martinfowler.com/eaaDev/Range.html.

[mf-ref-doc] Martin Fowler. “Refactoring Code to Load a Document.” https://martinfowler.com/articles/refactoring-document-load.html.

[mf-ref-pipe] Martin Fowler. “Refactoring with Loops and Collection Pipelines.” https://martinfowler.com/articles/refactoring-pipelines.html.

[mf-repos] Martin Fowler. “Repository.”

https://martinfowler.com/eaaCatalog/repository.html.

[mf-stc] Martin Fowler. “Bliki: SelfTestingCode.” https://martinfowler.com/bliki/SelfTestingCode.html.

[mf-tc] Martin Fowler. “Bliki: TestCoverage.” https://martinfowler.com/bliki/TestCoverage.html.

[mf-tdd] Martin Fowler. “Bliki: TestDrivenDevelopment.” https://martinfowler.com/bliki/TestDrivenDevelopment

[mf-tih] Martin Fowler. “Bliki: TypeInstanceHomonym.” https://martinfowler.com/bliki/TypeInstanceHomonym.

[mf-ua] Martin Fowler. “Bliki: UniformAccessPrinciple.” https://martinfowler.com/bliki/UniformAccessPrinciple

[mf-vo] Martin Fowler. “Bliki: ValueObject.” https://martinfowler.com/bliki/ValueObject.html.

[mf-xp] Martin Fowler. “Bliki: ExtremeProgramming.” https://martinfowler.com/bliki/ExtremeProgramming.ht

[mf-xunit] Martin Fowler. “Bliki: Xunit.”

https://martinfowler.com/bliki/Xunit.html.

[mf-yagni] Martin Fowler. “Bliki: Yagni.” https://martinfowler.com/bliki/Yagni.html.

[Opdyke] William F. Opdyke. “Refactoring Object-Oriented Frameworks.” Doctoral Dissertation. University of Illinois at Urbana-Champaign, 1992.

[Parnas] D.L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules.” In: *Communications of the ACM, Volume 15 Issue 12, pp. 1053–1058*. Dec. 1972.

[ref.com] https://refactoring.com.

[Wake] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003. ISBN 0321109295.

[wake-swap] Bill Wake.“The swap Statement Refactoring.”

1中译本：王海鹏等译，《数据库重构》，机械工业出版社，2007年，ISBN 9787111202097。——译者注

2中译本：刘未鹏译，《修改代码的艺术》，人民邮电出版社，2007年，ISBN 9787115163622。——译者注

3中译本：徐旭铭译，《重构（Ruby版）》，机械工业出版社，

2010年，ISBN 9787111300786。——译者注

4中译本：李英军等译，《设计模式》，机械工业出版社，2007年，ISBN 9787111211266。——译者注

5中译本：陈贤安等译，《重构HTML》，人民邮电出版社，2012年，ISBN 9787115294074。——译者注

6中译本：杨光、刘基诚译，《重构与模式》，人民邮电出版社，2013年，ISBN 9787115297259。——译者注

重构列表

Slide Statements	移动语句	223
Split Loop	拆分循环	227
Split Phase	拆分阶段	154
Split Variable	拆分变量	240
Substitute Algorithm	替换算法	195

坏味道与重构手法速查表

坏味道（英文）	坏味道（中文）	页码	常用重构
Alternative Classes with Different Interfaces	异曲同工的类	83	改变函数声明（124），搬移函数（198），提炼超类（375）
Comments	注释	84	提炼函数（106），改变函数声明（124），引入断言（302）
Data Class	纯数据类	83	封装记录（162），移除设值函数（331），搬移函数（198），提炼函数（106），拆分阶段（154）
Data Clumps	数据泥团	78	提炼类（182），引入参数对象（140），保持对象完整（319）

Divergent Change	发散式变化	76	拆分阶段（154），搬移函数（198），提炼函数（106），提炼类（182）
Duplicated Code	重复代码	72	提炼函数（106），移动语句（223），函数上移（350）
Feature Envy	依恋情结	77	搬移函数（198），提炼函数（106）
Global Data	全局数据	74	封装变量（132）
Insider Trading	内幕交易	82	搬移函数（198），搬移字段（207），隐藏委托关系（189），以委托取代子类（381），以委托取代超类（399）
Large Class	过大的类	82	提炼类（182），提炼超类（375），以子类取代类型码（362）
Lazy Element	冗赘的元素	80	内联函数（115），内联类（186），折叠继承体系（380）
Long Function	过长函数	73	提炼函数（106），以查询取代临时变量（178），引入参数对象（140），保持对象完整（319），以命令取代函数（337），分解条件表达式（260），以多态取代条件表达式（272），拆分循环（227）

Long Parameter List	过长参数列	74	以查询取代参数（324），保持对象完整（319），引入参数对象（140），移除标记参数（314），函数组合成类（144）
Loops	循环语句	79	以管道取代循环（231）
Message Chains	过长的消息链	81	隐藏委托关系（189），提炼函数（106），搬移函数（198）
Middle Man	中间人	81	移除中间人（192），内联函数（115），以委托取代超类（399），以委托取代子类（381）
Mutable Data	可变数据	75	封装变量（132），拆分变量（240），移动语句（223），提炼函数（106），将查询函数和修改函数分离（306），移除设值函数（331），以查询取代派生变量（248），函数组合成类（144），函数组合成变换（149），将引用对象改为值对象（252）
Mysterious Name	神秘命名	72	改变函数声明（124），变量改名（137），字段改名（244）
Primitive Obsession	基本类型偏执	78	以对象取代基本类型（174），以子类取代类型码（362），以多态取代条件表达式（272），提炼类（182），引入参数对象（140）
Refused	被拒	83	函数下移（359），字段下移（361），以委托取

Bequest	绝的遗赠		代子类（381），以委托取代超类（399）
Repeated Switches	重复的switch	79	以多态取代条件表达式（272）
Shotgun Surgery	霰弹式修改	76	搬移函数（198），搬移字段（207），函数组合成类（144），函数组合成变换（149），拆分阶段（154），内联函数（115），内联类（186）
Speculative Generality	夸夸其谈通用性	80	折叠继承体系（380），内联函数（115），内联类（186），改变函数声明（124），移除死代码（237）
Temporary Field	临时字段	80	提炼类（182），搬移函数（198），引入特例（289）

Table of Contents